# GNU Unifont

15.0.05

Generated by Doxygen 1.8.20

# Chapter 1

# Main Page

## 1.1 GNU Unifont C Utilities

This documentation covers C utility programs for creating GNU Unifont glyphs and fonts.

## 1.2 LICENSE



## 1.3 Introduction

Unifont is the creation of Roman Czyborra, who created Perl utilities for generating a dual-width Bitmap Distribution Format (BDF) font 16 pixels tall, unifont.bdf, from an input file named unifont.hex. The unifont.hex file contained two fields separated by a colon: a Unicode code point as four hexadecimal digits, and a hexadecimal string of 32 or 64 characters representing the glyph bitmap pattern. Roman also wrote other Perl scripts for manipulating unifont.hex files.

Jungshik Shin wrote a Perl script, johab2ucs2, to convert Hangul syllable glyph elements into Hangul Johab-encoded fonts. These glyph elements are compatible with Jaekyung "Jake" Song's Hanterm terminal emulator. Paul Hardy modified johab2ucs2 and drew Hangul Syllables Unicode elements for compatibility with this Johab encoding and with Hanterm. These new glyphs were created to avoid licensing issues with the Hangul Syllables glyphs that were in the original unifont.hex file.

Over time, Unifont was extended to allow correct positioning of combining marks in a TrueType font, coverage beyond Unicode Plane 0, and the addition of Under-ConScript Unicode Registry (UCSUR) glyphs. There is also partial support for experimental quadruple-width glyphs.

Paul Hardy wrote the first pair of C programs, unihex2bmp.c and unibmp2hex.c, to facilitate editing the bitmaps at their real aspect ratio. These programs allow conversion between the Unifont .hex format and a Windows Bitmap or Wireless Bitmap file for editing with a graphics editor. This was followed by make files, other C programs, Perl scripts, and shell scripts.

Luis Alejandro González Miranda wrote scripts for converting unifont.hex into a TrueType font using FontForge.

Andrew Miller wrote additional Perl programs for directly rendering unifont.hex files, for converting unifont.hex to and from Portable Network Graphics (PNG) files for editing based upon Paul Hardy's BMP conversion programs, and also wrote other Perl scripts.

David Corbett wrote a Perl script to rotate glyphs in a unifont.hex file and an awk script to substitute new glyphs for old glyphs of the same Unicode code point in a unifont.hex file.

**何志翔** (He Zhixiang) wrote a program to convert Unifont files into OpenType fonts, hex2otf.c.

## 1.4 The C Programs

This documentation only covers C programs and their header files. These programs are typically longer than the Unifont package's Perl scripts, which being much smaller are easier to understand. The C programs are, in alphabetical order:

| Program | Description |
|---|---|
| hex2otf.c | Convert a GNU Unifont .hex file to an OpenType font |
| unibdf2hex.c | Convert a BDF file into a unifont.hex file |
| unibmp2hex.c | Turn a .bmp or .wbmp glyph matrix into a GNU Unifont hex glyph set of 256 characters |
| unibmpbump.c | Adjust a Microsoft bitmap (.bmp) file that was created by unihex2png but converted to .bmp |
| unicoverage.c | Show the coverage of Unicode plane scripts for a GNU Unifont hex glyph file |
| unidup.c | Check for duplicate code points in sorted unifont.hex file |
| unifont1per.c | Read a Unifont .hex file from standard input and produce one glyph per .bmp bitmap file as output |
| unifontpic.c | See the "Big Picture": the entire Unifont in one BMP bitmap |
| unigencircles.c | Superimpose dashed combining circles on combining glyphs |
| unigenwidth.c | IEEE 1003.1-2008 setup to calculate wchar_t string widths |
| unihex2bmp.c | Turn a GNU Unifont hex glyph page of 256 code points into a bitmap for editing |
| unihexgen.c | Generate a series of glyphs containing hexadecimal code points |
| unipagecount.c | Count the number of glyphs defined in each page of 256 code points |

## 1.5 Perl Scripts

The very first program written for Unifont conversion was Roman Czyborra's hexdraw Perl script. That one script would convert a unifont.hex file into a text file with 16 lines per glyph (one for each glyph row) followed by a blank line aftr each glyph. That allowed editing unifont.hex glyphs with a text-based editor.

Combined with Roman's hex2bdf Perl script to convert a unifont.hex file into a BDF font, these two scripts formed a complete package for editing Unifont and generating the resulting BDF fonts.

There was no combining mark support initially, and the original unifont.hex file included combining circles with combining mark glyphs.

The list below gives a brief description of these and the other Perl scripts that are in the Unifont package src subdirectory.

| Perl Script | Description |
|---|---|
| bdfimplode | Convert a BDF font into GNU Unifont .hex format |
| hex2bdf | Convert a GNU Unifont .hex file into a BDF font |
| hex2sfd | Convert a GNU Unifont .hex file into a FontForge .sfd format |
| hexbraille | Algorithmically generate the Unicode Braille range (U+28xx) |
| hexdraw | Convert a GNU Unifont .hex file to and from an ASCII text file |
| hexkinya | Create the Private Use Area Kinya syllables |
| hexmerge | Merge two or more GNU Unifont .hex font files into one |
| johab2ucs2 | Convert a Johab BDF font into GNU Unifont Hangul Syllables |
| unifont-viewer | View a .hex font file with a graphical user interface |
| unifontchojung | Extract Hangul syllables that have no final consonant |
| unifontksx | Extract Hangul syllables that comprise KS X 1001:1992 |
| unihex2png | GNU Unifont .hex file to Portable Network Graphics converter |
| unihexfill | Generate range of Unifont 4- or 6-digit hexadecimal glyp |
| unihexrotate | Rotate Unifont hex glyphs in quarter turn increments |
| unipng2hex | Portable Network Graphics to GNU Unifont .hex file converter |

# Chapter 2

# Data Structure Index

## 2.1 Data Structures

Here are the data structures with brief descriptions:

# Chapter 3

# File Index

## 3.1 File List

Here is a list of all documented files with brief descriptions:

# Chapter 4

# Data Structure Documentation

## 4.1 Buffer Struct Reference

Generic data structure for a linked list of buffer elements.

### Data Fields

- size_t capacity
- byte * begin
- byte * next
- byte * end

### 4.1.1 Detailed Description

Generic data structure for a linked list of buffer elements.

A buffer can act as a vector (when filled with 'store*' functions), or a temporary output area (when filled with 'cache*' functions). The 'store*' functions use native endian. The 'cache*' functions use big endian or other formats in OpenType. Beware of memory alignment.

Definition at line 133 of file hex2otf.c.

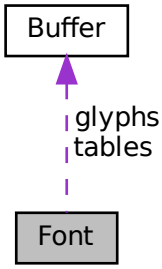
The documentation for this struct was generated from the following file:

- src/hex2otf.c

## 4.2 Font Struct Reference

Data structure to hold information for one font.

Collaboration diagram for Font:




### Data Fields

- Buffer * tables
- Buffer * glyphs
- uint_fast32_t glyphCount
- pixels_t maxWidth

### 4.2.1 Detailed Description

Data structure to hold information for one font.

Definition at line 628 of file hex2otf.c.

The documentation for this struct was generated from the following file:

- src/hex2otf.c

## 4.3 Glyph Struct Reference

Data structure to hold data for one bitmap glyph.

### Data Fields

- uint_least32_t codePoint
  undefined for glyph 0
- byte bitmap [GLYPH_MAX_BYTE_COUNT]
  hexadecimal bitmap character array
- uint_least8_t byteCount
  length of bitmap data
- bool combining
  whether this is a combining glyph
- pixels_t pos
- pixels_t lsb
  left side bearing (x position of leftmost contour point)

#### 4.3.1 Detailed Description

Data structure to hold data for one bitmap glyph.

This data structure holds data to represent one Unifont bitmap glyph: Unicode code point, number of bytes in its bitmap array, whether or not it is a combining character, and an offset from the glyph origin to the start of the bitmap.

Definition at line 614 of file hex2otf.c.

#### 4.3.2 Field Documentation

##### 4.3.2.1 pos

pixels_t Glyph::pos

number of pixels the glyph should be moved to the right (negative number means moving to the left)

Definition at line 620 of file hex2otf.c.

The documentation for this struct was generated from the following file:

- src/hex2otf.c

## 4.4 NamePair Struct Reference

Data structure for a font ID number and name character string.

#include <hex2otf.h>

### Data Fields

- int id
- const char * str

#### 4.4.1 Detailed Description

Data structure for a font ID number and name character string.

Definition at line 77 of file hex2otf.h.

The documentation for this struct was generated from the following file:

- src/hex2otf.h

## 4.5 Options Struct Reference

Data structure to hold options for OpenType font output.

### Data Fields

- bool truetype
- bool blankOutline
- bool bitmap
- bool gpos
- bool gsub
- int cff
- const char * hex
- const char * pos
- const char * out
- NameStrings nameStrings

#### 4.5.1 Detailed Description

Data structure to hold options for OpenType font output.

This data structure holds the status of options that can be specified as command line arguments for creating the output OpenType font file.

Definition at line 2453 of file hex2otf.c.

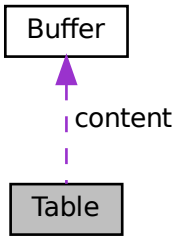
The documentation for this struct was generated from the following file:

- src/hex2otf.c

## 4.6 Table Struct Reference

Data structure for an OpenType table.

Collaboration diagram for Table:




### Data Fields

- uint_fast32_t tag
- Buffer * content

### 4.6.1 Detailed Description

Data structure for an OpenType table.

This data structure contains a table tag and a pointer to the start of the buffer that holds data for this OpenType table.

For information on the OpenType tables and their structure, see https://docs.microsoft.com/en-us/typography/opentype/spec/otff#font-tables.

Definition at line 645 of file hex2otf.c.

The documentation for this struct was generated from the following file:

- src/hex2otf.c

## 4.7 TableRecord Struct Reference

Data structure for data associated with one OpenType table.

### Data Fields

- uint_least32_t tag
- uint_least32_t offset
- uint_least32_t length
- uint_least32_t checksum

### 4.7.1 Detailed Description

Data structure for data associated with one OpenType table.

This data structure contains an OpenType table's tag, start within an OpenType font file, length in bytes, and checksum at the end of the table.

Definition at line 747 of file hex2otf.c.

The documentation for this struct was generated from the following file:

- src/hex2otf.c

# Chapter 5

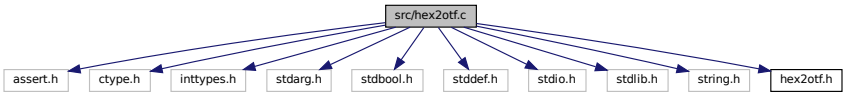
# File Documentation

## 5.1 src/hex2otf.c File Reference

hex2otf - Convert GNU Unifont .hex file to OpenType font

```
#include <assert.h>
#include <ctype.h>
#include <inttypes.h>
#include <stdarg.h>
#include <stdbool.h>
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "hex2otf.h"
```

Include dependency graph for hex2otf.c:




### Data Structures

- struct Buffer

  Generic data structure for a linked list of buffer elements.
- struct Glyph

  Data structure to hold data for one bitmap glyph.
- struct Font

  Data structure to hold information for one font.
- struct Table

  Data structure for an OpenType table.
- struct TableRecord

  Data structure for data associated with one OpenType table.
- struct Options

  Data structure to hold options for OpenType font output.

## Macros

- #define VERSION "1.0.1"

  Program version, for "--version" option.
- #define U16MAX 0xffff

  Maximum UTF-16 code point value.
- #define U32MAX 0xffffffff

  Maximum UTF-32 code point value.
- #define PRI_CP "U+%.4"PRIXFAST32

  Format string to print Unicode code point.
- #define static_assert(a, b) (assert(a))

  If "a" is true, return string "b".
- #define BX(shift, x) ((uintmax_t)(!!(x)) << (shift))

  Truncate & shift word.
- #define B0(shift) BX((shift), 0)

  Clear a given bit in a word.
- #define B1(shift) BX((shift), 1)

  Set a given bit in a word.
- #define GLYPH_MAX_WIDTH 16

  Maximum glyph width, in pixels.
- #define GLYPH_HEIGHT 16

  Maximum glyph height, in pixels.
- #define GLYPH_MAX_BYTE_COUNT (GLYPH_HEIGHT * GLYPH_MAX_WIDTH / 8)

  Number of bytes to represent one bitmap glyph as a binary array.
- #define DESCENDER 2

  Count of pixels below baseline.
- #define ASCENDER (GLYPH_HEIGHT - DESCENDER)

  Count of pixels above baseline.
- #define FUPEM 64

  Font units per em.
- #define MAX_GLYPHS 65536

  An OpenType font has at most 65536 glyphs.
- #define MAX_NAME_IDS 256

  Name IDs 0-255 are used for standard names.
- #define FU(x) ((x) * FUPEM / GLYPH_HEIGHT)

  Convert pixels to font units.
- #define PW(x) ((x) / (GLYPH_HEIGHT / 8))

  Convert glyph byte count to pixel width.
- #define defineStore(name, type)

  Temporary define to look up an element in an array of given type.
- #define addByte(shift)
- #define getRowBit(rows, x, y) ((rows)[(y)] & x0 >> (x))
- #define flipRowBit(rows, x, y) ((rows)[(y)] ^= x0 >> (x))
- #define stringCount (sizeof strings / sizeof *strings)
- #define cacheCFF32(buf, x) (cacheU8 ((buf), 29), cacheU32 ((buf), (x)))

## Typedefs

- typedef unsigned char byte
  Definition of ”byte” type as an unsigned char.
- typedef int_least8_t pixels_t
  This type must be able to represent max(GLYPH_MAX_WIDTH, GLYPH_HEIGHT).
- typedef struct Buffer Buffer
  Generic data structure for a linked list of buffer elements.
- typedef const char * NameStrings[MAX_NAME_IDS]
  Array of OpenType names indexed directly by Name IDs.
- typedef struct Glyph Glyph
  Data structure to hold data for one bitmap glyph.
- typedef struct Font Font
  Data structure to hold information for one font.
- typedef struct Table Table
  Data structure for an OpenType table.
- typedef struct Options Options
  Data structure to hold options for OpenType font output.

## Enumerations

- enum LocaFormat { LOCA_OFFSET16 = 0, LOCA_OFFSET32 = 1 }
  Index to Location (”loca”) offset information.
- enum ContourOp { OP_CLOSE, OP_POINT }
  Specify the current contour drawing operation.
- enum FillSide { FILL_LEFT, FILL_RIGHT }
  Fill to the left side (CFF) or right side (TrueType) of a contour.

## Functions

- void fail (const char *reason,...)
  Print an error message on stderr, then exit.
- void initBuffers (size_t count)
  Initialize an array of buffer pointers to all zeroes.
- void cleanBuffers ()
  Free all allocated buffer pointers.
- Buffer * newBuffer (size_t initialCapacity)
  Create a new buffer.
- void ensureBuffer (Buffer *buf, size_t needed)
  Ensure that the buffer has at least the specified minimum size.
- void freeBuffer (Buffer *buf)
  Free the memory previously allocated for a buffer.
- defineStore (storeU8, uint_least8_t)
- void cacheU8 (Buffer *buf, uint_fast8_t value)
  Append one unsigned byte to the end of a byte array.
- void cacheU16 (Buffer *buf, uint_fast16_t value)

Append two unsigned bytes to the end of a byte array.

- void cacheU32 (Buffer *buf, uint_fast32_t value)

  Append four unsigned bytes to the end of a byte array.
- void cacheCFFOperand (Buffer *buf, int_fast32_t value)

  Cache charstring number encoding in a CFF buffer.
- void cacheZeros (Buffer *buf, size_t count)

  Append 1 to 4 bytes of zeroes to a buffer, for padding.
- void cacheBytes (Buffer *restrict buf, const void *restrict src, size_t count)

  Append a string of bytes to a buffer.
- void cacheBuffer (Buffer *restrict bufDest, const Buffer *restrict bufSrc)

  Append bytes of a table to a byte buffer.
- void writeBytes (const byte bytes[], size_t count, FILE *file)

  Write an array of bytes to an output file.
- void writeU16 (uint_fast16_t value, FILE *file)

  Write an unsigned 16-bit value to an output file.
- void writeU32 (uint_fast32_t value, FILE *file)

  Write an unsigned 32-bit value to an output file.
- void addTable (Font *font, const char tag[static 4], Buffer *content)

  Add a TrueType or OpenType table to the font.
- void organizeTables (Font *font, bool isCFF)

  Sort tables according to OpenType recommendations.
- int byTableTag (const void *a, const void *b)

  Compare tables by 4-byte unsigned table tag value.
- void writeFont (Font *font, bool isCFF, const char *fileName)

  Write OpenType font to output file.
- bool readCodePoint (uint_fast32_t *codePoint, const char *fileName, FILE *file)

  Read up to 6 hexadecimal digits and a colon from file.
- void readGlyphs (Font *font, const char *fileName)

  Read glyph definitions from a Unifont .hex format file.
- int byCodePoint (const void *a, const void *b)

  Compare two Unicode code points to determine which is greater.
- void positionGlyphs (Font *font, const char *fileName, pixels_t *xMin)

  Position a glyph within a 16-by-16 pixel bounding box.
- void sortGlyphs (Font *font)

  Sort the glyphs in a font by Unicode code point.
- void buildOutline (Buffer *result, const byte bitmap[], const size_t byteCount, const enum FillSide fillSide)

  Build a glyph outline.
- void prepareOffsets (size_t *sizes)

  Prepare 32-bit glyph offsets in a font table.
- Buffer * prepareStringIndex (const NameStrings names)

  Prepare a font name string index.
- void fillCFF (Font *font, int version, const NameStrings names)

  Add a CFF table to a font.
- void fillTrueType (Font *font, enum LocaFormat *format, uint_fast16_t *maxPoints, uint_fast16_t *maxContours)

  Add a TrueType table to a font.

- void fillBlankOutline (Font *font)

  Create a dummy blank outline in a font table.
- void fillBitmap (Font *font)

  Fill OpenType bitmap data and location tables.
- void fillHeadTable (Font *font, enum LocaFormat locaFormat, pixels_t xMin)

  Fill a "head" font table.
- void fillHheaTable (Font *font, pixels_t xMin)

  Fill a "hhea" font table.
- void fillMaxpTable (Font *font, bool isCFF, uint_fast16_t maxPoints, uint_fast16_t maxContours)

  Fill a "maxp" font table.
- void fillOS2Table (Font *font)

  Fill an "OS/2" font table.
- void fillHmtxTable (Font *font)

  Fill an "hmtx" font table.
- void fillCmapTable (Font *font)

  Fill a "cmap" font table.
- void fillPostTable (Font *font)

  Fill a "post" font table.
- void fillGposTable (Font *font)

  Fill a "GPOS" font table.
- void fillGsubTable (Font *font)

  Fill a "GSUB" font table.
- void cacheStringAsUTF16BE (Buffer *buf, const char *str)

  Cache a string as a big-ending UTF-16 surrogate pair.
- void fillNameTable (Font *font, NameStrings nameStrings)

  Fill a "name" font table.
- void printVersion ()

  Print program version string on stdout.
- void printHelp ()

  Print help message to stdout and then exit.
- const char * matchToken (const char *operand, const char *key, char delimiter)

  Match a command line option with its key for enabling.
- Options parseOptions (char *const argv[const])

  Parse command line options.
- int main (int argc, char *argv[])

  The main function.

## Variables

- Buffer * allBuffers

  Initial allocation of empty array of buffer pointers.
- size_t bufferCount

  Number of buffers in a Buffer * array.
- size_t nextBufferIndex

  Index number to tail element of Buffer * array.

### 5.1.1 Detailed Description

hex2otf - Convert GNU Unifont .hex file to OpenType font

This program reads a Unifont .hex format file and a file containing combining mark offset information, and produces an OpenType font file.



Author

**何志翔** (He Zhixiang)

### 5.1.2 Macro Definition Documentation

#### 5.1.2.1 addByte

```
#define addByte(
                    shift )
```

Value:

```
if (p == end) \
    break; \
record->checksum += (uint_fast32_t)*p++ « (shift);
```

#### 5.1.2.2 defineStore

```
#define defineStore(
                    name,
                    type )
```

Value:

```
void name (Buffer *buf, type value) \
{ \
   type *slot = getBufferSlot (buf, sizeof value); \
   *slot = value; \
}
```

Temporary define to look up an element in an array of given type.

This defintion is used to create lookup functions to return a given element in unsigned arrays of size 8, 16, and 32 bytes, and in an array of pixels.

Definition at line 350 of file hex2otf.c.

### 5.1.3 Typedef Documentation

#### 5.1.3.1 Buffer

typedef struct Buffer Buffer

Generic data structure for a linked list of buffer elements.

A buffer can act as a vector (when filled with 'store∗' functions), or a temporary output area (when filled with 'cache∗' functions). The 'store∗' functions use native endian. The 'cache∗' functions use big endian or other formats in OpenType. Beware of memory alignment.

#### 5.1.3.2 Glyph

typedef struct Glyph Glyph

Data structure to hold data for one bitmap glyph.

This data structure holds data to represent one Unifont bitmap glyph: Unicode code point, number of bytes in its bitmap array, whether or not it is a combining character, and an offset from the glyph origin to the start of the bitmap.

#### 5.1.3.3 Options

typedef struct Options Options

Data structure to hold options for OpenType font output.

This data structure holds the status of options that can be specified as command line arguments for creating the output OpenType font file.

#### 5.1.3.4 Table

typedef struct Table Table

Data structure for an OpenType table.

This data structure contains a table tag and a pointer to the start of the buffer that holds data for this OpenType table.

For information on the OpenType tables and their structure, see https://docs.microsoft.com/en-us/typography/opentype/spec/otff#font-tables.

### 5.1.4 Enumeration Type Documentation

#### 5.1.4.1 ContourOp

enum ContourOp

Specify the current contour drawing operation.

Enumerator

| | |
|---|---|
| OP_CLOSE | Close the current contour path that was being drawn. |
| OP_POINT | Add one more (x,y) point to the contor being drawn. |

Definition at line 1136 of file hex2otf.c.

```
1136             {
1137     OP_CLOSE,    ///< Close the current contour path that was being drawn.
1138     OP_POINT     ///< Add one more (x,y) point to the contor being drawn.
1139 };
```

#### 5.1.4.2 FillSide

enum FillSide

Fill to the left side (CFF) or right side (TrueType) of a contour.

Enumerator

| | |
|---|---|
| FILL_LEFT | Draw outline counter-clockwise (CFF, PostScript). |
| FILL_RIGHT | Draw outline clockwise (TrueType). |

Definition at line 1144 of file hex2otf.c.

```
1144             {
1145     FILL_LEFT,    ///< Draw outline counter-clockwise (CFF, PostScript).
1146     FILL_RIGHT    ///< Draw outline clockwise (TrueType).
1147 };
```

#### 5.1.4.3 LocaFormat

enum LocaFormat

Index to Location ("loca") offset information.

This enumerated type encodes the type of offset to locations in a table. It denotes Offset16 (16-bit) and Offset32 (32-bit) offset types.

Enumerator

| | |
|---|---|
| LOCA_OFFSET16 | Offset to location is a 16-bit Offset16 value. |
| LOCA_OFFSET32 | Offset to location is a 32-bit Offset32 value. |

Definition at line 658 of file hex2otf.c.

```
658             {
```

```
659     LOCA_OFFSET16 = 0,    ///< Offset to location is a 16-bit Offset16 value
660     LOCA_OFFSET32 = 1     ///< Offset to location is a 32-bit Offset32 value
661 };
```

### 5.1.5 Function Documentation

#### 5.1.5.1 addTable()

```
void addTable (
                Font * font,
                const char tag[static 4],
                Buffer * content )
```

Add a TrueType or OpenType table to the font.

This function adds a TrueType or OpenType table to a font. The 4-byte table tag is passed as an unsigned 32-bit integer in big-endian format.

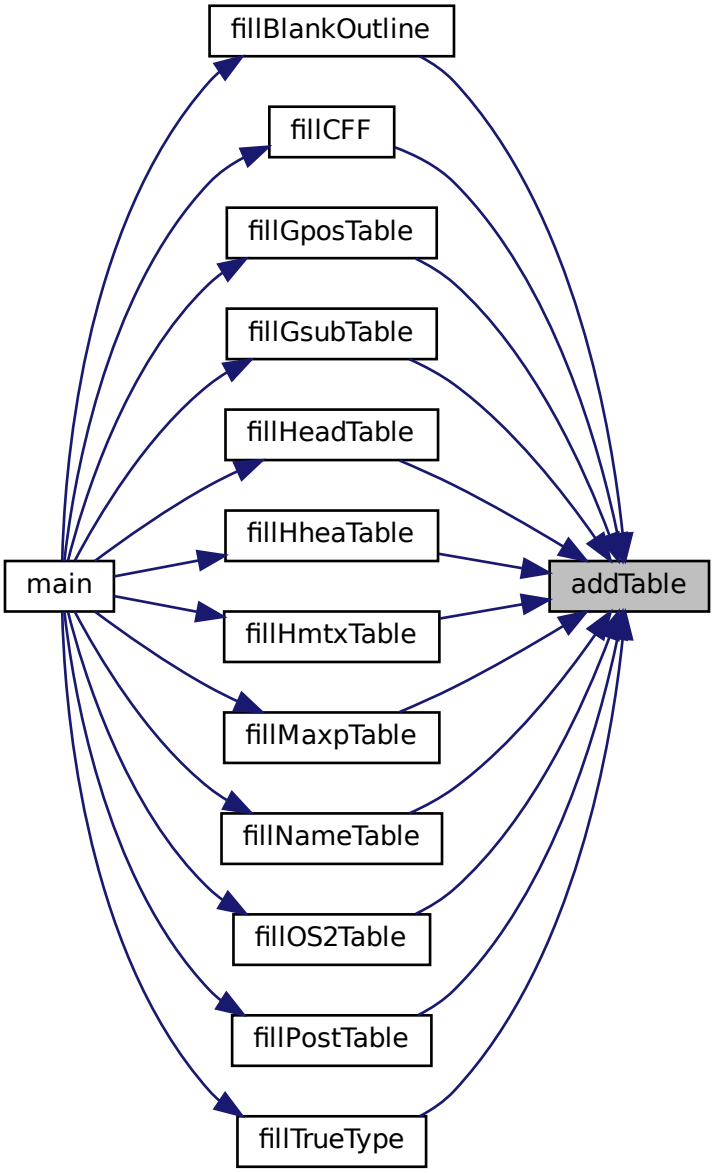
Parameters

| | | |
|---|---|---|
| in,out | font | The font to which a font table will be added. |
| in | tag | The 4-byte table name. |
| in | content | The table bytes to add, of type Buffer *. |

Definition at line 694 of file hex2otf.c.

```
695 {
696     Table *table = getBufferSlot (font->tables, sizeof (Table));
697     table->tag = tagAsU32 (tag);
698     table->content = content;
699 }
```

Here is the caller graph for this function:




#### 5.1.5.2 buildOutline()

void buildOutline (
Buffer * result,

```
const byte bitmap[],
const size_t byteCount,
const enum FillSide fillSide )
```

Build a glyph outline.

This function builds a glyph outline from a Unifont glyph bitmap.

Parameters

| | | |
|---|---|---|
| out | result | The resulting glyph outline. |
| in | bitmap | A bitmap array. |
| in | byteCount | the number of bytes in the input bitmap array. |
| in | fillSide | Enumerated indicator to fill left or right side. |

Get the value of a given bit that is in a given row.

Invert the value of a given bit that is in a given row.

Definition at line 1160 of file hex2otf.c.

```
1162 {
1163     enum Direction {RIGHT, LEFT, DOWN, UP}; // order is significant
1164
1165     // respective coordinate deltas
1166     const pixels_t dx[] = {1, -1, 0, 0}, dy[] = {0, 0, -1, 1};
1167
1168     assert (byteCount % GLYPH_HEIGHT == 0);
1169     const uint_fast8_t bytesPerRow = byteCount / GLYPH_HEIGHT;
1170     const pixels_t glyphWidth = bytesPerRow * 8;
1171     assert (glyphWidth <= GLYPH_MAX_WIDTH);
1172
1173 #if GLYPH_MAX_WIDTH < 32
1174         typedef uint_fast32_t row_t;
1175 #elif GLYPH_MAX_WIDTH < 64
1176         typedef uint_fast64_t row_t;
1177 #else
1178 #error GLYPH_MAX_WIDTH is too large.
1179 #endif
1180
1181     row_t pixels[GLYPH_HEIGHT + 2] = {0};
1182     for (pixels_t row = GLYPH_HEIGHT; row > 0; row--)
1183         for (pixels_t b = 0; b < bytesPerRow; b++)
1184             pixels[row] = pixels[row] « 8 | *bitmap++;
1185     typedef row_t graph_t[GLYPH_HEIGHT + 1];
1186     graph_t vectors[4];
1187     const row_t *lower = pixels, *upper = pixels + 1;
1188     for (pixels_t row = 0; row <= GLYPH_HEIGHT; row++)
1189     {
1190         const row_t m = (fillSide == FILL_RIGHT) - 1;
1191         vectors[RIGHT][row] = (m ^ (*lower « 1)) & (~m ^ (*upper « 1));
1192         vectors[LEFT ][row] = (m ^ (*upper     )) & (~m ^ (*lower     ));
1193         vectors[DOWN ][row] = (m ^ (*lower     )) & (~m ^ (*lower « 1));
1194         vectors[UP   ][row] = (m ^ (*upper « 1)) & (~m ^ (*upper     ));
1195         lower++;
1196         upper++;
1197     }
1198     graph_t selection = {0};
1199     const row_t x0 = (row_t)1 « glyphWidth;
1200
1201 /// Get the value of a given bit that is in a given row.
1202 #define getRowBit(rows, x, y)  ((rows)[(y)] &  x0 » (x))
1203
1204 /// Invert the value of a given bit that is in a given row.
1205 #define flipRowBit(rows, x, y) ((rows)[(y)] ^= x0 » (x))
1206
1207     for (pixels_t y = GLYPH_HEIGHT; y >= 0; y--)
1208     {
1209         for (pixels_t x = 0; x <= glyphWidth; x++)
```

```
1210        {
1211            assert (!getRowBit (vectors[LEFT], x, y));
1212            assert (!getRowBit (vectors[UP], x, y));
1213            enum Direction initial;
1214
1215            if (getRowBit (vectors[RIGHT], x, y))
1216                initial = RIGHT;
1217            else if (getRowBit (vectors[DOWN], x, y))
1218                initial = DOWN;
1219            else
1220                continue;
1221
1222            static_assert ((GLYPH_MAX_WIDTH + 1) * (GLYPH_HEIGHT + 1) * 2 <=
1223                U16MAX, "potential overflow");
1224
1225            uint_fast16_t lastPointCount = 0;
1226            for (bool converged = false;;)
1227            {
1228                uint_fast16_t pointCount = 0;
1229                enum Direction heading = initial;
1230                for (pixels_t tx = x, ty = y;;)
1231                {
1232                    if (converged)
1233                    {
1234                        storePixels (result, OP_POINT);
1235                        storePixels (result, tx);
1236                        storePixels (result, ty);
1237                    }
1238                    do
1239                    {
1240                        if (converged)
1241                            flipRowBit (vectors[heading], tx, ty);
1242                        tx += dx[heading];
1243                        ty += dy[heading];
1244                    } while (getRowBit (vectors[heading], tx, ty));
1245                    if (tx == x && ty == y)
1246                        break;
1247                    static_assert ((UP ^ DOWN) == 1 && (LEFT ^ RIGHT) == 1,
1248                        "wrong enums");
1249                    heading = (heading & 2) ^ 2;
1250                    heading |= !!getRowBit (selection, tx, ty);
1251                    heading ^= !getRowBit (vectors[heading], tx, ty);
1252                    assert (getRowBit (vectors[heading], tx, ty));
1253                    flipRowBit (selection, tx, ty);
1254                    pointCount++;
1255                }
1256                if (converged)
1257                    break;
1258                converged = pointCount == lastPointCount;
1259                lastPointCount = pointCount;
1260            }
1261
1262            storePixels (result, OP_CLOSE);
1263        }
1264    }
1265 #undef getRowBit
1266 #undef flipRowBit
1267 }
```

#### 5.1.5.3 byCodePoint()

```
int byCodePoint (
                const void * a,
                const void * b )
```

Compare two Unicode code points to determine which is greater.

This function compares the Unicode code points contained within two Glyph data structures. The function returns 1 if the first code point is greater, and -1 if the second is greater.

Parameters

| | | |
|---|---|---|
| in | a | A Glyph data structure containing the first code point. |
| in | b | A Glyph data structure containing the second code point. |

Returns

1 if the code point a is greater, -1 if less, 0 if equal.

Definition at line 1040 of file hex2otf.c.

```
1041 {
1042     const Glyph *const ga = a, *const gb = b;
1043     int gt = ga->codePoint > gb->codePoint;
1044     int lt = ga->codePoint < gb->codePoint;
1045     return gt - lt;
1046 }
```

#### 5.1.5.4 byTableTag()

```
int byTableTag (
                const void * a,
                const void * b )
```

Compare tables by 4-byte unsigned table tag value.

This function takes two pointers to a TableRecord data structure and extracts the four-byte tag structure element for each. The two 32-bit numbers are then compared. If the first tag is greater than the first, then gt = 1 and lt = 0, and so 1 - 0 = 1 is returned. If the first is less than the second, then gt = 0 and lt = 1, and so 0 - 1 = -1 is returned.

Parameters

| | | |
|---|---|---|
| in | a | Pointer to the first TableRecord structure. |
| in | b | Pointer to the second TableRecord structure. |

Returns

1 if the tag in "a" is greater, -1 if less, 0 if equal.

Definition at line 767 of file hex2otf.c.

```
768 {
769     const struct TableRecord *const ra = a, *const rb = b;
770     int gt = ra->tag > rb->tag;
771     int lt = ra->tag < rb->tag;
772     return gt - lt;
773 }
```

#### 5.1.5.5 cacheBuffer()

```
void cacheBuffer (
                Buffer *restrict bufDest,
                const Buffer *restrict bufSrc )
```

Append bytes of a table to a byte buffer.

Parameters

| in,out | bufDest | The buffer to which the new bytes are appended. |
|---|---|---|
| in | bufSrc | The bytes to append to the buffer array. |

Definition at line 523 of file hex2otf.c.

```
524 {
525     size_t length = countBufferedBytes (bufSrc);
526     ensureBuffer (bufDest, length);
527     memcpy (bufDest->next, bufSrc->begin, length);
528     bufDest->next += length;
529 }
```

#### 5.1.5.6 cacheBytes()

```
void cacheBytes (
                Buffer *restrict buf,
                const void *restrict src,
                size_t count )
```

Append a string of bytes to a buffer.

This function appends an array of 1 to 4 bytes to the end of a buffer.

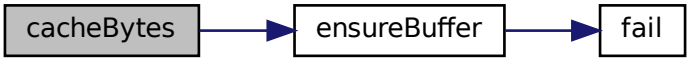
Parameters

| in,out | buf | The buffer to which the bytes are appended. |
|---|---|---|
| in | src | The array of bytes to append to the buffer. |
| in | count | The number of bytes containing zeroes to append. |

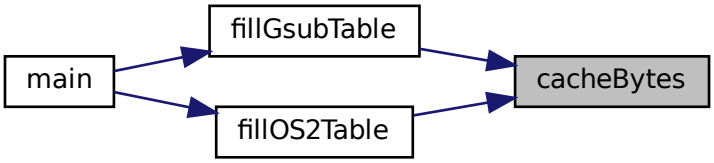
Definition at line 509 of file hex2otf.c.

```
510 {
511     ensureBuffer (buf, count);
512     memcpy (buf->next, src, count);
513     buf->next += count;
514 }
```

Here is the call graph for this function:




Here is the caller graph for this function:




#### 5.1.5.7 cacheCFFOperand()

```
void cacheCFFOperand (
                Buffer * buf,
                int_fast32_t value )
```

Cache charstring number encoding in a CFF buffer.

This function caches two's complement 8-, 16-, and 32-bit words as per Adobe's Type 2 Charstring encoding for operands. These operands are used in Compact Font Format data structures.

Byte values can have offsets, for which this function compensates, optionally followed by additional bytes:

| Byte Range | Offset | Bytes | Adjusted Range |
|---|---|---|---|
| 0 to 11 | 0 | 1 | 0 to 11 (operators) |
| 12 | 0 | 2 | Next byte is 8-bit op code |
| 13 to 18 | 0 | 1 | 13 to 18 (operators) |
| 19 to 20 | 0 | 2+ | hintmask and cntrmask operators |
| 21 to 27 | 0 | 1 | 21 to 27 (operators) |
| 28 | 0 | 3 | 16-bit 2's complement number |
| 29 to 31 | 0 | 1 | 29 to 31 (operators) |
| 32 to 246 | -139 | 1 | -107 to +107 |
| 247 to 250 | +108 | 2 | +108 to +1131 |
| 251 to 254 | -108 | 2 | -108 to -1131 |
| 255 | 0 | 5 | 16-bit integer and 16-bit fraction |

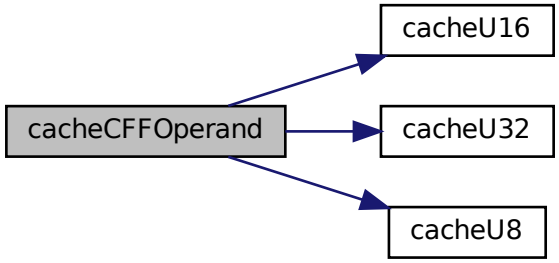
Parameters

| | | |
|---|---|---|
| in,out | buf | The buffer to which the operand value is appended. |
| in | value | The operand value. |

Definition at line 460 of file hex2otf.c.

```
461 {
462     if (-107 <= value && value <= 107)
463         cacheU8 (buf, value + 139);
464     else if (108 <= value && value <= 1131)
465     {
466         cacheU8 (buf, (value - 108) / 256 + 247);
467         cacheU8 (buf, (value - 108) % 256);
468     }
469     else if (-32768 <= value && value <= 32767)
470     {
471         cacheU8 (buf, 28);
472         cacheU16 (buf, value);
473     }
474     else if (-2147483647 <= value && value <= 2147483647)
475     {
476         cacheU8 (buf, 29);
477         cacheU32 (buf, value);
478     }
479     else
480         assert (false); // other encodings are not used and omitted
481     static_assert (GLYPH_MAX_WIDTH <= 107, "More encodings are needed.");
482 }
```

Here is the call graph for this function:




### 5.1.5.8 cacheStringAsUTF16BE()

```
void cacheStringAsUTF16BE (
                Buffer * buf,
                const char * str )
```

Cache a string as a big-ending UTF-16 surrogate pair.

This function encodes a UTF-8 string as a big-endian UTF-16 surrogate pair.

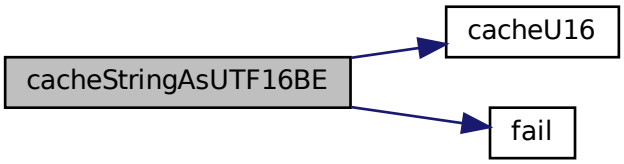
Parameters

| | | |
|---|---|---|
| in,out | buf | Pointer to a Buffer struct to update. |
| in | str | The character array to encode. |

Definition at line 2316 of file hex2otf.c.

```
2317 {
2318     for (const char *p = str; *p; p++)
2319     {
2320         byte c = *p;
2321         if (c < 0x80)
2322         {
2323             cacheU16 (buf, c);
2324             continue;
2325         }
2326         int length = 1;
2327         byte mask = 0x40;
2328         for (; c & mask; mask »= 1)
2329             length++;
2330         if (length == 1 || length > 4)
2331             fail ("Ill-formed UTF-8 sequence.");
2332         uint_fast32_t codePoint = c & (mask - 1);
2333         for (int i = 1; i < length; i++)
2334         {
2335             c = *++p;
2336             if ((c & 0xc0) != 0x80) // NUL checked here
2337                 fail ("Ill-formed UTF-8 sequence.");
2338             codePoint = (codePoint « 6) | (c & 0x3f);
2339         }
2340         const int lowerBits = length==2 ? 7 : length==3 ? 11 : 16;
2341         if (codePoint » lowerBits == 0)
2342             fail ("Ill-formed UTF-8 sequence."); // sequence should be shorter
2343         if (codePoint >= 0xd800 && codePoint <= 0xdfff)
2344             fail ("Ill-formed UTF-8 sequence.");
2345         if (codePoint > 0x10ffff)
2346             fail ("Ill-formed UTF-8 sequence.");
2347         if (codePoint > 0xffff)
2348         {
2349             cacheU16 (buf, 0xd800 | (codePoint - 0x10000) » 10);
2350             cacheU16 (buf, 0xdc00 | (codePoint & 0x3ff));
2351         }
2352         else
2353             cacheU16 (buf, codePoint);
2354     }
2355 }
```

Here is the call graph for this function:



#### 5.1.5.9 cacheU16()

```
void cacheU16 (
            Buffer * buf,
            uint_fast16_t value )
```

Append two unsigned bytes to the end of a byte array.

This function adds two bytes to the end of a byte array. The buffer is updated to account for the newly-added bytes.

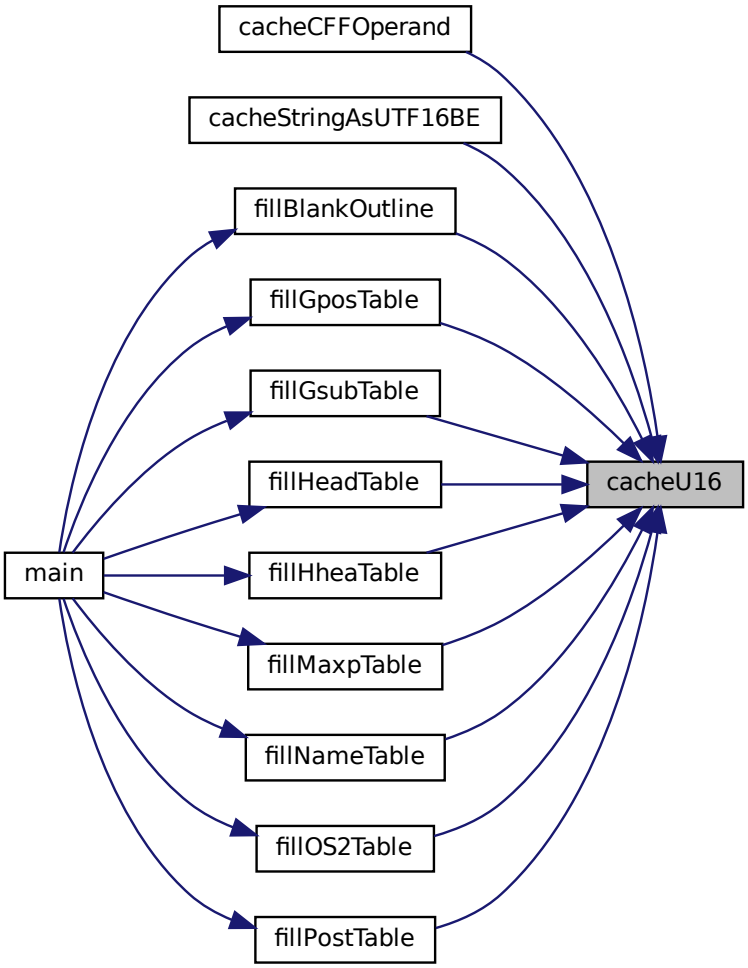
Parameters

| | | |
|---|---|---|
| in,out | buf | The array of bytes to which to append two new bytes. |
| in | value | The 16-bit unsigned value to append to the buf array. |

Definition at line 412 of file hex2otf.c.

```
413 {
414     cacheU (buf, value, 2);
415 }
```

Here is the caller graph for this function:




#### 5.1.5.10 cacheU32()

```
void cacheU32 (
            Buffer * buf,
            uint_fast32_t value )
```

Append four unsigned bytes to the end of a byte array.

This function adds four bytes to the end of a byte array. The buffer is updated to account for the newly-added bytes.

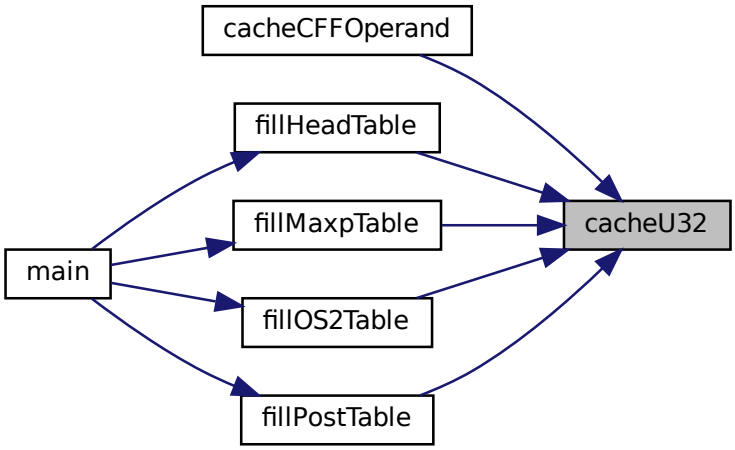
Parameters

| | | |
|---|---|---|
| in,out | buf | The array of bytes to which to append four new bytes. |
| in | value | The 32-bit unsigned value to append to the buf array. |

Definition at line 427 of file hex2otf.c.

```
428 {
429     cacheU (buf, value, 4);
430 }
```

Here is the caller graph for this function:




#### 5.1.5.11 cacheU8()

```
void cacheU8 (
                Buffer * buf,
                uint_fast8_t value )
```

Append one unsigned byte to the end of a byte array.

This function adds one byte to the end of a byte array. The buffer is updated to account for the newly-added byte.

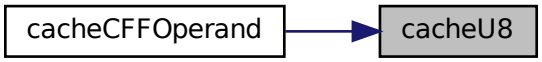
Parameters

| | | |
|---|---|---|
| in,out | buf | The array of bytes to which to append a new byte. |
| in | value | The 8-bit unsigned value to append to the buf array. |

Definition at line 397 of file hex2otf.c.

```
398 {
399     storeU8 (buf, value & 0xff);
400 }
```

Here is the caller graph for this function:




#### 5.1.5.12 cacheZeros()

```
void cacheZeros (
                Buffer * buf,
                size_t count )
```

Append 1 to 4 bytes of zeroes to a buffer, for padding.

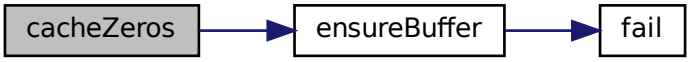
Parameters

| | | |
|---|---|---|
| in,out | buf | The buffer to which the operand value is appended. |
| in | count | The number of bytes containing zeroes to append. |

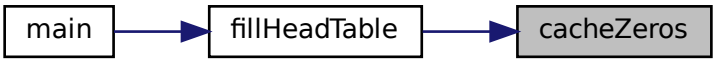
Definition at line 491 of file hex2otf.c.

```
492 {
493     ensureBuffer (buf, count);
494     memset (buf->next, 0, count);
495     buf->next += count;
496 }
```

Here is the call graph for this function:



Here is the caller graph for this function:




#### 5.1.5.13 cleanBuffers()

void cleanBuffers ( )

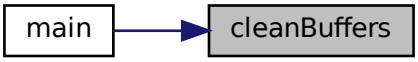
Free all allocated buffer pointers.

This function frees all buffer pointers previously allocated in the initBuffers function.

Definition at line 170 of file hex2otf.c.

```
171 {
172     for (size_t i = 0; i < bufferCount; i++)
173         if (allBuffers[i].capacity)
174             free (allBuffers[i].begin);
175     free (allBuffers);
176     bufferCount = 0;
177 }
```

Here is the caller graph for this function:




#### 5.1.5.14 ensureBuffer()

void ensureBuffer (
Buffer * buf,
size_t needed )

Ensure that the buffer has at least the specified minimum size.

This function takes a buffer array of type Buffer and the necessary minimum number of elements as inputs, and attempts to increase the size of the buffer if it must be larger.

If the buffer is too small and cannot be resized, the program will terminate with an error message and an exit status of EXIT_FAILURE.

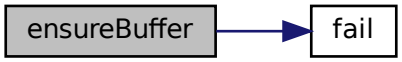
Parameters

| | | |
|---|---|---|
| in,out | buf | The buffer to check. |
| in | needed | The required minimum number of elements in the buffer. |

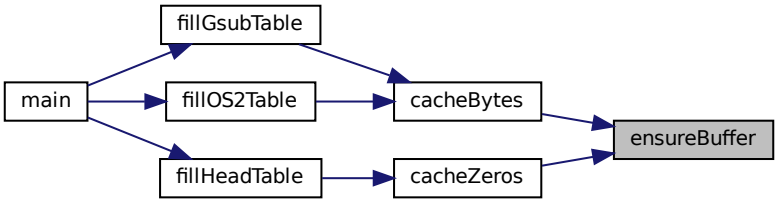
Definition at line 239 of file hex2otf.c.

```
240 {
241     if (buf->end - buf->next >= needed)
242         return;
243     ptrdiff_t occupied = buf->next - buf->begin;
244     size_t required = occupied + needed;
245     if (required < needed) // overflow
246         fail ("Cannot allocate %zu + %zu bytes of memory.", occupied, needed);
247     if (required > SIZE_MAX / 2)
248         buf->capacity = required;
249     else while (buf->capacity < required)
250         buf->capacity *= 2;
251     void *extended = realloc (buf->begin, buf->capacity);
252     if (!extended)
253         fail ("Failed to allocate %zu bytes of memory.", buf->capacity);
254     buf->begin = extended;
255     buf->next = buf->begin + occupied;
256     buf->end = buf->begin + buf->capacity;
257 }
```

Here is the call graph for this function:




Here is the caller graph for this function:



#### 5.1.5.15 fail()

```
void fail (
            const char * reason,
            ... )
```

Print an error message on stderr, then exit.

This function prints the provided error string and optional following arguments to stderr, and then exits with a status of EXIT_FAILURE.

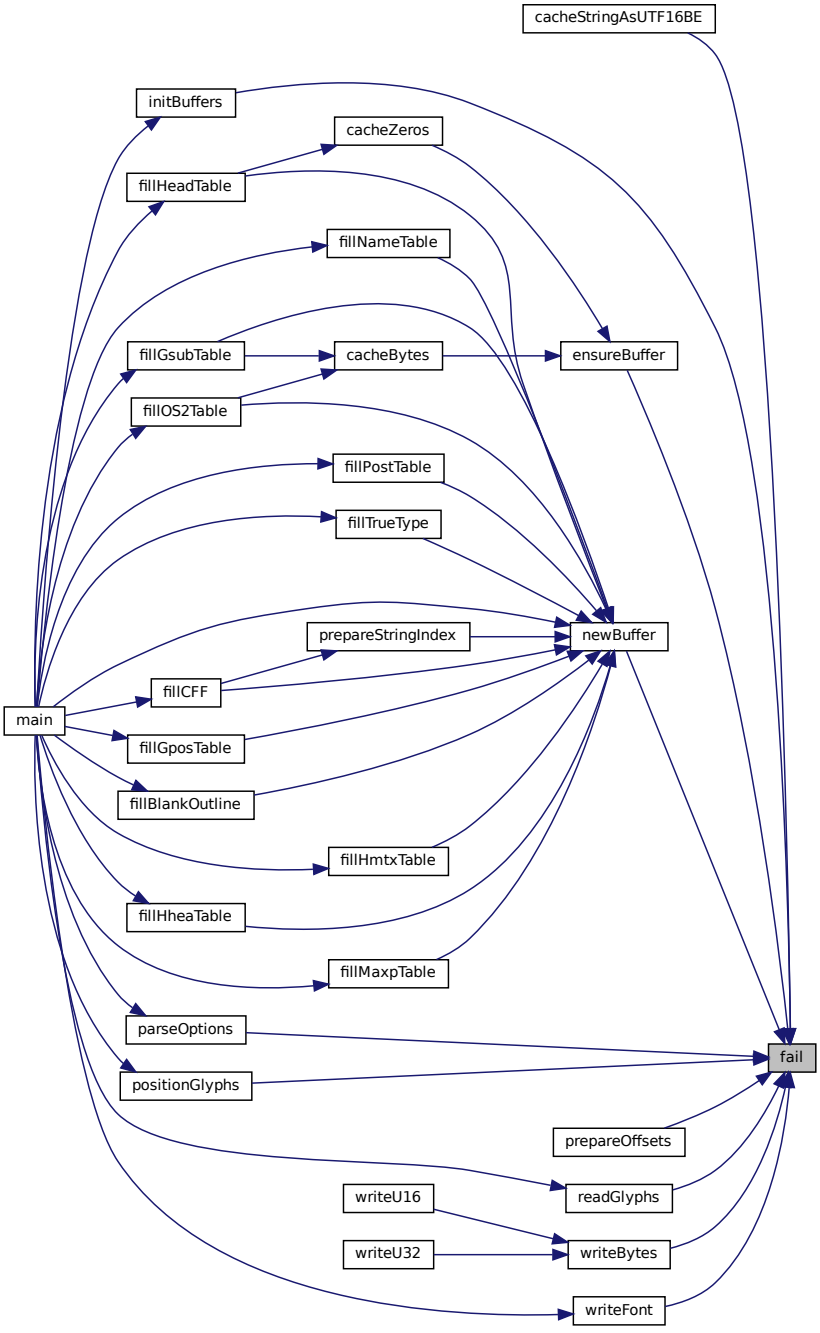
Parameters

| | | |
|---|---|---|
| in | reason | The output string to describe the error. |
| in | ... | Optional following arguments to output. |

Definition at line 113 of file hex2otf.c.

```
114 {
115     fputs ("ERROR: ", stderr);
116     va_list args;
117     va_start (args, reason);
118     vfprintf (stderr, reason, args);
119     va_end (args);
120     putc ('\n', stderr);
121     exit (EXIT_FAILURE);
122 }
```

Here is the caller graph for this function:



#### 5.1.5.16 fillBitmap()

void fillBitmap (
Font * font )

Fill OpenType bitmap data and location tables.

This function fills an Embedded Bitmap Data (EBDT) Table and an Embedded Bitmap Location (EBLC) Table with glyph bitmap information. These tables enable embedding bitmaps in OpenType fonts. No Embedded Bitmap Scaling (EBSC) table is used for the bitmap glyphs, only EBDT and EBLC.

Parameters

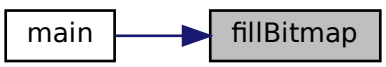
| | | |
|---|---|---|
| in,out | font | Pointer to a Font struct in which to add bitmaps. |

Definition at line 1728 of file hex2otf.c.

```
1729 {
1730     const Glyph *const glyphs = getBufferHead (font->glyphs);
1731     const Glyph *const glyphsEnd = getBufferTail (font->glyphs);
1732     size_t bitmapsSize = 0;
1733     for (const Glyph *glyph = glyphs; glyph < glyphsEnd; glyph++)
1734         bitmapsSize += glyph->byteCount;
1735     Buffer *ebdt = newBuffer (4 + bitmapsSize);
1736     addTable (font, "EBDT", ebdt);
1737     cacheU16 (ebdt, 2); // majorVersion
1738     cacheU16 (ebdt, 0); // minorVersion
1739     uint_fast8_t byteCount = 0; // unequal to any glyph
1740     pixels_t pos = 0;
1741     bool combining = false;
1742     Buffer *rangeHeads = newBuffer (32);
1743     Buffer *offsets = newBuffer (64);
1744     for (const Glyph *glyph = glyphs; glyph < glyphsEnd; glyph++)
1745     {
1746         if (glyph->byteCount != byteCount || glyph->pos != pos ||
1747             glyph->combining != combining)
1748         {
1749             storeU16 (rangeHeads, glyph - glyphs);
1750             storeU32 (offsets, countBufferedBytes (ebdt));
1751             byteCount = glyph->byteCount;
1752             pos = glyph->pos;
1753             combining = glyph->combining;
1754         }
1755         cacheBytes (ebdt, glyph->bitmap, byteCount);
1756     }
1757     const uint_least16_t *ranges = getBufferHead (rangeHeads);
1758     const uint_least16_t *rangesEnd = getBufferTail (rangeHeads);
1759     uint_fast32_t rangeCount = rangesEnd - ranges;
1760     storeU16 (rangeHeads, font->glyphCount);
1761     Buffer *eblc = newBuffer (4096);
1762     addTable (font, "EBLC", eblc);
1763     cacheU16 (eblc, 2); // majorVersion
1764     cacheU16 (eblc, 0); // minorVersion
1765     cacheU32 (eblc, 1); // numSizes
1766     { // bitmapSizes[0]
1767         cacheU32 (eblc, 56); // indexSubTableArrayOffset
1768         cacheU32 (eblc, (8 + 20) * rangeCount); // indexTablesSize
1769         cacheU32 (eblc, rangeCount); // numberOfIndexSubTables
1770         cacheU32 (eblc, 0); // colorRef
1771         { // hori
1772             cacheU8 (eblc, ASCENDER); // ascender
1773             cacheU8 (eblc, -DESCENDER); // descender
1774             cacheU8 (eblc, font->maxWidth); // widthMax
1775             cacheU8 (eblc, 1); // caretSlopeNumerator
1776             cacheU8 (eblc, 0); // caretSlopeDenominator
1777             cacheU8 (eblc, 0); // caretOffset
1778             cacheU8 (eblc, 0); // minOriginSB
1779             cacheU8 (eblc, 0); // minAdvanceSB
1780             cacheU8 (eblc, ASCENDER); // maxBeforeBL
1781             cacheU8 (eblc, -DESCENDER); // minAfterBL
1782             cacheU8 (eblc, 0); // pad1
```

```
1783            cacheU8 (eblc, 0); // pad2
1784        }
1785        { // vert
1786            cacheU8 (eblc, ASCENDER); // ascender
1787            cacheU8 (eblc, -DESCENDER); // descender
1788            cacheU8 (eblc, font->maxWidth); // widthMax
1789            cacheU8 (eblc, 1); // caretSlopeNumerator
1790            cacheU8 (eblc, 0); // caretSlopeDenominator
1791            cacheU8 (eblc, 0); // caretOffset
1792            cacheU8 (eblc, 0); // minOriginSB
1793            cacheU8 (eblc, 0); // minAdvanceSB
1794            cacheU8 (eblc, ASCENDER); // maxBeforeBL
1795            cacheU8 (eblc, -DESCENDER); // minAfterBL
1796            cacheU8 (eblc, 0); // pad1
1797            cacheU8 (eblc, 0); // pad2
1798        }
1799        cacheU16 (eblc, 0); // startGlyphIndex
1800        cacheU16 (eblc, font->glyphCount - 1); // endGlyphIndex
1801        cacheU8 (eblc, 16); // ppemX
1802        cacheU8 (eblc, 16); // ppemY
1803        cacheU8 (eblc, 1); // bitDepth
1804        cacheU8 (eblc, 1); // flags = Horizontal
1805    }
1806    { // IndexSubTableArray
1807        uint_fast32_t offset = rangeCount * 8;
1808        for (const uint_least16_t *p = ranges; p < rangesEnd; p++)
1809        {
1810            cacheU16 (eblc, *p); // firstGlyphIndex
1811            cacheU16 (eblc, p[1] - 1); // lastGlyphIndex
1812            cacheU32 (eblc, offset); // additionalOffsetToIndexSubtable
1813            offset += 20;
1814        }
1815    }
1816    { // IndexSubTables
1817        const uint_least32_t *offset = getBufferHead (offsets);
1818        for (const uint_least16_t *p = ranges; p < rangesEnd; p++)
1819        {
1820            const Glyph *glyph = &glyphs[*p];
1821            cacheU16 (eblc, 2); // indexFormat
1822            cacheU16 (eblc, 5); // imageFormat
1823            cacheU32 (eblc, *offset++); // imageDataOffset
1824            cacheU32 (eblc, glyph->byteCount); // imageSize
1825            { // bigMetrics
1826                cacheU8 (eblc, GLYPH_HEIGHT); // height
1827                const uint_fast8_t width = PW (glyph->byteCount);
1828                cacheU8 (eblc, width); // width
1829                cacheU8 (eblc, glyph->pos); // horiBearingX
1830                cacheU8 (eblc, ASCENDER); // horiBearingY
1831                cacheU8 (eblc, glyph->combining ? 0 : width); // horiAdvance
1832                cacheU8 (eblc, 0); // vertBearingX
1833                cacheU8 (eblc, 0); // vertBearingY
1834                cacheU8 (eblc, GLYPH_HEIGHT); // vertAdvance
1835            }
1836        }
1837    }
1838    freeBuffer (rangeHeads);
1839    freeBuffer (offsets);
1840 }
```

Here is the caller graph for this function:



#### 5.1.5.17 fillBlankOutline()

void fillBlankOutline (
Font * font )

Create a dummy blank outline in a font table.

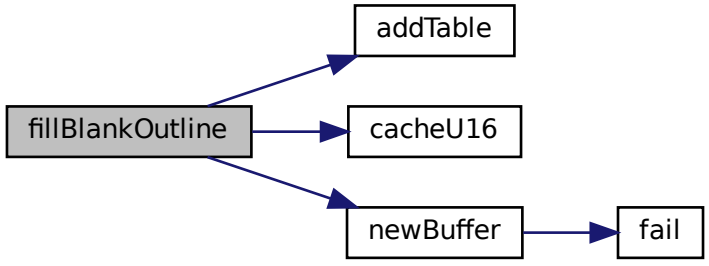
Parameters

| | | |
|---|---|---|
| in,out | font | Pointer to a Font struct to insert a blank outline. |

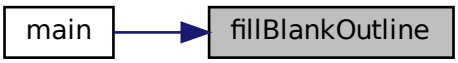
Definition at line 1697 of file hex2otf.c.

```
1698 {
1699     Buffer *glyf = newBuffer (12);
1700     addTable (font, "glyf", glyf);
1701     // Empty table is not allowed, but an empty outline for glyph 0 suffices.
1702     cacheU16 (glyf, 0); // numberOfContours
1703     cacheU16 (glyf, FU (0)); // xMin
1704     cacheU16 (glyf, FU (0)); // yMin
1705     cacheU16 (glyf, FU (0)); // xMax
1706     cacheU16 (glyf, FU (0)); // yMax
1707     cacheU16 (glyf, 0); // instructionLength
1708     Buffer *loca = newBuffer (2 * (font->glyphCount + 1));
1709     addTable (font, "loca", loca);
1710     cacheU16 (loca, 0); // offsets[0]
1711     assert (countBufferedBytes (glyf) % 2 == 0);
1712     for (uint_fast32_t i = 1; i <= font->glyphCount; i++)
1713         cacheU16 (loca, countBufferedBytes (glyf) / 2); // offsets[i]
1714 }
```

Here is the call graph for this function:



Here is the caller graph for this function:




#### 5.1.5.18 fillCFF()

```
void fillCFF (
        Font * font,
        int version,
        const NameStrings names )
```

Add a CFF table to a font.

Parameters

| | | |
|---|---|---|
| in,out | font | Pointer to a Font struct to contain the CFF table. |
| in | version | Version of CFF table, with value 1 or 2. |
| in | names | List of NameStrings. |

Use fixed width integer for variables to simplify offset calculation.

Definition at line 1329 of file hex2otf.c.

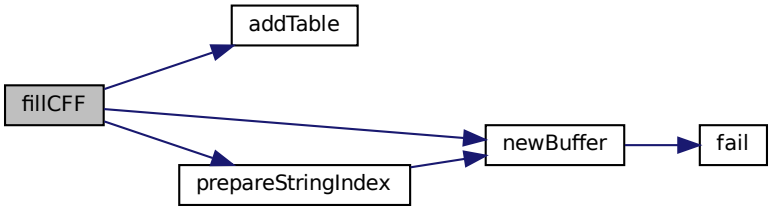
```
1330 {
1331     // HACK: For convenience, CFF data structures are hard coded.
1332     assert (0 < version && version <= 2);
1333     Buffer *cff = newBuffer (65536);
1334     addTable (font, version == 1 ? "CFF " : "CFF2", cff);
1335
1336 /// Use fixed width integer for variables to simplify offset calculation.
1337 #define cacheCFF32(buf, x) (cacheU8 ((buf), 29), cacheU32 ((buf), (x)))
1338
1339     // In Unifont, 16px glyphs are more common. This is used by CFF1 only.
1340     const pixels_t defaultWidth = 16, nominalWidth = 8;
1341     if (version == 1)
1342     {
1343         Buffer *strings = prepareStringIndex (names);
1344         size_t stringsSize = countBufferedBytes (strings);
1345         const char *cffName = names[6];
1346         assert (cffName);
1347         size_t nameLength = strlen (cffName);
1348         size_t namesSize = nameLength + 5;
1349         // These sizes must be updated together with the data below.
1350         size_t offsets[] = {4, namesSize, 45, stringsSize, 2, 5, 8, 32, 4, 0};
1351         prepareOffsets (offsets);
1352         { // Header
1353             cacheU8 (cff, 1); // major
1354             cacheU8 (cff, 0); // minor
1355             cacheU8 (cff, 4); // hdrSize
```

```
1356            cacheU8 (cff, 1); // offSize
1357        }
1358        assert (countBufferedBytes (cff) == offsets[0]);
1359        { // Name INDEX (should not be used by OpenType readers)
1360            cacheU16 (cff, 1); // count
1361            cacheU8 (cff, 1); // offSize
1362            cacheU8 (cff, 1); // offset[0]
1363            if (nameLength + 1 > 255) // must be too long; spec limit is 63
1364                fail ("PostScript name is too long.");
1365            cacheU8 (cff, nameLength + 1); // offset[1]
1366            cacheBytes (cff, cffName, nameLength);
1367        }
1368        assert (countBufferedBytes (cff) == offsets[1]);
1369        { // Top DICT INDEX
1370            cacheU16 (cff, 1); // count
1371            cacheU8 (cff, 1); // offSize
1372            cacheU8 (cff, 1); // offset[0]
1373            cacheU8 (cff, 41); // offset[1]
1374            cacheCFFOperand (cff, 391); // "Adobe"
1375            cacheCFFOperand (cff, 392); // "Identity"
1376            cacheCFFOperand (cff, 0);
1377            cacheBytes (cff, (byte[]){12, 30}, 2); // ROS
1378            cacheCFF32 (cff, font->glyphCount);
1379            cacheBytes (cff, (byte[]){12, 34}, 2); // CIDCount
1380            cacheCFF32 (cff, offsets[6]);
1381            cacheBytes (cff, (byte[]){12, 36}, 2); // FDArray
1382            cacheCFF32 (cff, offsets[5]);
1383            cacheBytes (cff, (byte[]){12, 37}, 2); // FDSelect
1384            cacheCFF32 (cff, offsets[4]);
1385            cacheU8 (cff, 15); // charset
1386            cacheCFF32 (cff, offsets[8]);
1387            cacheU8 (cff, 17); // CharStrings
1388        }
1389        assert (countBufferedBytes (cff) == offsets[2]);
1390        { // String INDEX
1391            cacheBuffer (cff, strings);
1392            freeBuffer (strings);
1393        }
1394        assert (countBufferedBytes (cff) == offsets[3]);
1395        cacheU16 (cff, 0); // Global Subr INDEX
1396        assert (countBufferedBytes (cff) == offsets[4]);
1397        { // Charsets
1398            cacheU8 (cff, 2); // format
1399            { // Range2[0]
1400                cacheU16 (cff, 1); // first
1401                cacheU16 (cff, font->glyphCount - 2); // nLeft
1402            }
1403        }
1404        assert (countBufferedBytes (cff) == offsets[5]);
1405        { // FDSelect
1406            cacheU8 (cff, 3); // format
1407            cacheU16 (cff, 1); // nRanges
1408            cacheU16 (cff, 0); // first
1409            cacheU8 (cff, 0); // fd
1410            cacheU16 (cff, font->glyphCount); // sentinel
1411        }
1412        assert (countBufferedBytes (cff) == offsets[6]);
1413        { // FDArray
1414            cacheU16 (cff, 1); // count
1415            cacheU8 (cff, 1); // offSize
1416            cacheU8 (cff, 1); // offset[0]
1417            cacheU8 (cff, 28); // offset[1]
1418            cacheCFFOperand (cff, 393);
1419            cacheBytes (cff, (byte[]){12, 38}, 2); // FontName
1420            // Windows requires FontMatrix in Font DICT.
1421            const byte unit[] = {0x1e,0x15,0x62,0x5c,0x6f}; // 1/64 (0.015625)
1422            cacheBytes (cff, unit, sizeof unit);
1423            cacheCFFOperand (cff, 0);
1424            cacheCFFOperand (cff, 0);
1425            cacheBytes (cff, unit, sizeof unit);
1426            cacheCFFOperand (cff, 0);
1427            cacheCFFOperand (cff, 0);
1428            cacheBytes (cff, (byte[]){12, 7}, 2); // FontMatrix
1429            cacheCFFOperand (cff, offsets[8] - offsets[7]); // size
1430            cacheCFF32 (cff, offsets[7]); // offset
1431            cacheU8 (cff, 18); // Private
1432        }
1433        assert (countBufferedBytes (cff) == offsets[7]);
1434        { // Private
1435            cacheCFFOperand (cff, FU (defaultWidth));
1436            cacheU8 (cff, 20); // defaultWidthX
```

```
1437            cacheCFFOperand (cff, FU (nominalWidth));
1438            cacheU8 (cff, 21); // nominalWidthX
1439        }
1440        assert (countBufferedBytes (cff) == offsets[8]);
1441    }
1442    else
1443    {
1444        assert (version == 2);
1445        // These sizes must be updated together with the data below.
1446        size_t offsets[] = {5, 21, 4, 10, 0};
1447        prepareOffsets (offsets);
1448        { // Header
1449            cacheU8 (cff, 2); // majorVersion
1450            cacheU8 (cff, 0); // minorVersion
1451            cacheU8 (cff, 5); // headerSize
1452            cacheU16 (cff, offsets[1] - offsets[0]); // topDictLength
1453        }
1454        assert (countBufferedBytes (cff) == offsets[0]);
1455        { // Top DICT
1456            const byte unit[] = {0x1e,0x15,0x62,0x5c,0x6f}; // 1/64 (0.015625)
1457            cacheBytes (cff, unit, sizeof unit);
1458            cacheCFFOperand (cff, 0);
1459            cacheCFFOperand (cff, 0);
1460            cacheBytes (cff, unit, sizeof unit);
1461            cacheCFFOperand (cff, 0);
1462            cacheCFFOperand (cff, 0);
1463            cacheBytes (cff, (byte[]){12, 7}, 2); // FontMatrix
1464            cacheCFFOperand (cff, offsets[2]);
1465            cacheBytes (cff, (byte[]){12, 36}, 2); // FDArray
1466            cacheCFFOperand (cff, offsets[3]);
1467            cacheU8 (cff, 17); // CharStrings
1468        }
1469        assert (countBufferedBytes (cff) == offsets[1]);
1470        cacheU32 (cff, 0); // Global Subr INDEX
1471        assert (countBufferedBytes (cff) == offsets[2]);
1472        { // Font DICT INDEX
1473            cacheU32 (cff, 1); // count
1474            cacheU8 (cff, 1); // offSize
1475            cacheU8 (cff, 1); // offset[0]
1476            cacheU8 (cff, 4); // offset[1]
1477            cacheCFFOperand (cff, 0);
1478            cacheCFFOperand (cff, 0);
1479            cacheU8 (cff, 18); // Private
1480        }
1481        assert (countBufferedBytes (cff) == offsets[3]);
1482    }
1483    { // CharStrings INDEX
1484        Buffer *offsets = newBuffer (4096);
1485        Buffer *charstrings = newBuffer (4096);
1486        Buffer *outline = newBuffer (1024);
1487        const Glyph *glyph = getBufferHead (font->glyphs);
1488        const Glyph *const endGlyph = glyph + font->glyphCount;
1489        for (; glyph < endGlyph; glyph++)
1490        {
1491            // CFF offsets start at 1
1492            storeU32 (offsets, countBufferedBytes (charstrings) + 1);
1493
1494            pixels_t rx = -glyph->pos;
1495            pixels_t ry = DESCENDER;
1496            resetBuffer (outline);
1497            buildOutline (outline, glyph->bitmap, glyph->byteCount, FILL_LEFT);
1498            enum CFFOp {rmoveto=21, hmoveto=22, vmoveto=4, hlineto=6,
1499                vlineto=7, endchar=14};
1500            enum CFFOp pendingOp = 0;
1501            const int STACK_LIMIT = version == 1 ? 48 : 513;
1502            int stackSize = 0;
1503            bool isDrawing = false;
1504            pixels_t width = glyph->combining ? 0 : PW (glyph->byteCount);
1505            if (version == 1 && width != defaultWidth)
1506            {
1507                cacheCFFOperand (charstrings, FU (width - nominalWidth));
1508                stackSize++;
1509            }
1510            for (const pixels_t *p = getBufferHead (outline),
1511                 *const end = getBufferTail (outline); p < end;)
1512            {
1513                int s = 0;
1514                const enum ContourOp op = *p++;
1515                if (op == OP_POINT)
1516                {
1517                    const pixels_t x = *p++, y = *p++;
```

```
1518                 if (x != rx)
1519                 {
1520                     cacheCFFOperand (charstrings, FU (x - rx));
1521                     rx = x;
1522                     stackSize++;
1523                     s |= 1;
1524                 }
1525                 if (y != ry)
1526                 {
1527                     cacheCFFOperand (charstrings, FU (y - ry));
1528                     ry = y;
1529                     stackSize++;
1530                     s |= 2;
1531                 }
1532                 assert (!(isDrawing && s == 3));
1533             }
1534             if (s)
1535             {
1536                 if (!isDrawing)
1537                 {
1538                     const enum CFFOp moves[] = {0, hmoveto, vmoveto,
1539                         rmoveto};
1540                     cacheU8 (charstrings, moves[s]);
1541                     stackSize = 0;
1542                 }
1543                 else if (!pendingOp)
1544                     pendingOp = (enum CFFOp[]){0, hlineto, vlineto}[s];
1545             }
1546             else if (!isDrawing)
1547             {
1548                 // only when the first point happens to be (0, 0)
1549                 cacheCFFOperand (charstrings, FU (0));
1550                 cacheU8 (charstrings, hmoveto);
1551                 stackSize = 0;
1552             }
1553             if (op == OP_CLOSE || stackSize >= STACK_LIMIT)
1554             {
1555                 assert (stackSize <= STACK_LIMIT);
1556                 cacheU8 (charstrings, pendingOp);
1557                 pendingOp = 0;
1558                 stackSize = 0;
1559             }
1560             isDrawing = op != OP_CLOSE;
1561         }
1562         if (version == 1)
1563             cacheU8 (charstrings, endchar);
1564     }
1565     size_t lastOffset = countBufferedBytes (charstrings) + 1;
1566 #if SIZE_MAX > U32MAX
1567         if (lastOffset > U32MAX)
1568             fail ("CFF data exceeded size limit.");
1569 #endif
1570     storeU32 (offsets, lastOffset);
1571     int offsetSize = 1 + (lastOffset > 0xff)
1572                        + (lastOffset > 0xffff)
1573                        + (lastOffset > 0xffffff);
1574     // count (must match 'numGlyphs' in 'maxp' table)
1575     cacheU (cff, font->glyphCount, version * 2);
1576     cacheU8 (cff, offsetSize); // offSize
1577     const uint_least32_t *p = getBufferHead (offsets);
1578     const uint_least32_t *const end = getBufferTail (offsets);
1579     for (; p < end; p++)
1580         cacheU (cff, *p, offsetSize); // offsets
1581     cacheBuffer (cff, charstrings); // data
1582     freeBuffer (offsets);
1583     freeBuffer (charstrings);
1584     freeBuffer (outline);
1585   }
1586 #undef cacheCFF32
1587 }
```

Here is the call graph for this function:




Here is the caller graph for this function:


#### 5.1.5.19 fillCmapTable()

void fillCmapTable (
Font * font )

Fill a ”cmap” font table.

The ”cmap” table contains character to glyph index mapping information.

Parameters

| | | |
|---|---|---|
| in,out | font | The Font struct to which to add the table. |

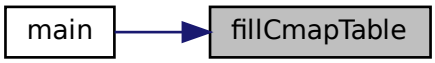
Definition at line 2109 of file hex2otf.c.

```
2110 {
2111     Glyph *const glyphs = getBufferHead (font->glyphs);
2112     Buffer *rangeHeads = newBuffer (16);
2113     uint_fast32_t rangeCount = 0;
2114     uint_fast32_t bmpRangeCount = 1; // 1 for the last 0xffff-0xffff range
2115     glyphs[0].codePoint = glyphs[1].codePoint; // to start a range at glyph 1
2116     for (uint_fast16_t i = 1; i < font->glyphCount; i++)
```

```
2117    {
2118        if (glyphs[i].codePoint != glyphs[i - 1].codePoint + 1)
2119        {
2120            storeU16 (rangeHeads, i);
2121            rangeCount++;
2122            bmpRangeCount += glyphs[i].codePoint < 0xffff;
2123        }
2124    }
2125    Buffer *cmap = newBuffer (256);
2126    addTable (font, "cmap", cmap);
2127    // Format 4 table is always generated for compatibility.
2128    bool hasFormat12 = glyphs[font->glyphCount - 1].codePoint > 0xffff;
2129    cacheU16 (cmap, 0); // version
2130    cacheU16 (cmap, 1 + hasFormat12); // numTables
2131    { // encodingRecords[0]
2132        cacheU16 (cmap, 3); // platformID
2133        cacheU16 (cmap, 1); // encodingID
2134        cacheU32 (cmap, 12 + 8 * hasFormat12); // subtableOffset
2135    }
2136    if (hasFormat12) // encodingRecords[1]
2137    {
2138        cacheU16 (cmap, 3); // platformID
2139        cacheU16 (cmap, 10); // encodingID
2140        cacheU32 (cmap, 36 + 8 * bmpRangeCount); // subtableOffset
2141    }
2142    const uint_least16_t *ranges = getBufferHead (rangeHeads);
2143    const uint_least16_t *const rangesEnd = getBufferTail (rangeHeads);
2144    storeU16 (rangeHeads, font->glyphCount);
2145    { // format 4 table
2146        cacheU16 (cmap, 4); // format
2147        cacheU16 (cmap, 16 + 8 * bmpRangeCount); // length
2148        cacheU16 (cmap, 0); // language
2149        if (bmpRangeCount * 2 > U16MAX)
2150            fail ("Too many ranges in 'cmap' table.");
2151        cacheU16 (cmap, bmpRangeCount * 2); // segCountX2
2152        uint_fast16_t searchRange = 1, entrySelector = -1;
2153        while (searchRange <= bmpRangeCount)
2154        {
2155            searchRange «= 1;
2156            entrySelector++;
2157        }
2158        cacheU16 (cmap, searchRange); // searchRange
2159        cacheU16 (cmap, entrySelector); // entrySelector
2160        cacheU16 (cmap, bmpRangeCount * 2 - searchRange); // rangeShift
2161        { // endCode[]
2162            const uint_least16_t *p = ranges;
2163            for (p++; p < rangesEnd && glyphs[*p].codePoint < 0xffff; p++)
2164                cacheU16 (cmap, glyphs[*p - 1].codePoint);
2165            uint_fast32_t cp = glyphs[*p - 1].codePoint;
2166            if (cp > 0xfffe)
2167                cp = 0xfffe;
2168            cacheU16 (cmap, cp);
2169            cacheU16 (cmap, 0xffff);
2170        }
2171        cacheU16 (cmap, 0); // reservedPad
2172        { // startCode[]
2173            for (uint_fast32_t i = 0; i < bmpRangeCount - 1; i++)
2174                cacheU16 (cmap, glyphs[ranges[i]].codePoint);
2175            cacheU16 (cmap, 0xffff);
2176        }
2177        { // idDelta[]
2178            const uint_least16_t *p = ranges;
2179            for (; p < rangesEnd && glyphs[*p].codePoint < 0xffff; p++)
2180                cacheU16 (cmap, *p - glyphs[*p].codePoint);
2181            uint_fast16_t delta = 1;
2182            if (p < rangesEnd && *p == 0xffff)
2183                delta = *p - glyphs[*p].codePoint;
2184            cacheU16 (cmap, delta);
2185        }
2186        { // idRangeOffsets[]
2187            for (uint_least16_t i = 0; i < bmpRangeCount; i++)
2188                cacheU16 (cmap, 0);
2189        }
2190    }
2191    if (hasFormat12) // format 12 table
2192    {
2193        cacheU16 (cmap, 12); // format
2194        cacheU16 (cmap, 0); // reserved
2195        cacheU32 (cmap, 16 + 12 * rangeCount); // length
2196        cacheU32 (cmap, 0); // language
2197        cacheU32 (cmap, rangeCount); // numGroups
```

```
2198
2199         // groups[]
2200         for (const uint_least16_t *p = ranges; p < rangesEnd; p++)
2201         {
2202             cacheU32 (cmap, glyphs[*p].codePoint); // startCharCode
2203             cacheU32 (cmap, glyphs[p[1] - 1].codePoint); // endCharCode
2204             cacheU32 (cmap, *p); // startGlyphID
2205         }
2206     }
2207     freeBuffer (rangeHeads);
2208 }
```

Here is the caller graph for this function:




#### 5.1.5.20 fillGposTable()

void fillGposTable (
Font * font )

Fill a "GPOS" font table.

The "GPOS" table contains information for glyph positioning.

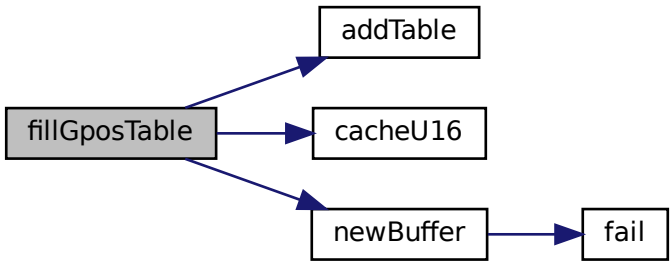
Parameters

| | | |
|---|---|---|
| in,out | font | The Font struct to which to add the table. |

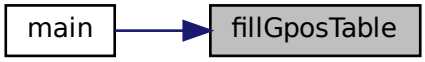
Definition at line 2241 of file hex2otf.c.

```
2242 {
2243     Buffer *gpos = newBuffer (16);
2244     addTable (font, "GPOS", gpos);
2245     cacheU16 (gpos, 1); // majorVersion
2246     cacheU16 (gpos, 0); // minorVersion
2247     cacheU16 (gpos, 10); // scriptListOffset
2248     cacheU16 (gpos, 12); // featureListOffset
2249     cacheU16 (gpos, 14); // lookupListOffset
2250     { // ScriptList table
2251         cacheU16 (gpos, 0); // scriptCount
2252     }
2253     { // Feature List table
2254         cacheU16 (gpos, 0); // featureCount
2255     }
2256     { // Lookup List Table
2257         cacheU16 (gpos, 0); // lookupCount
2258     }
2259 }
```

Here is the call graph for this function:




Here is the caller graph for this function:




#### 5.1.5.21 fillGsubTable()

```
void fillGsubTable (
                Font * font )
```

Fill a ”GSUB” font table.

The ”GSUB” table contains information for glyph substitution.

Parameters

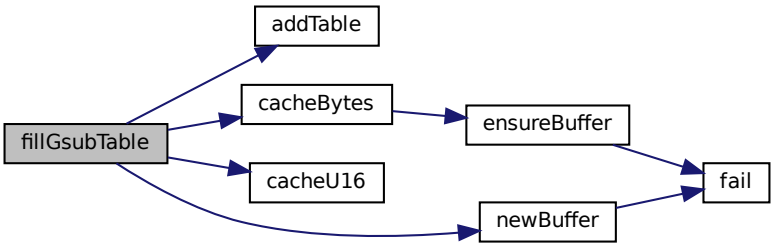
| | | |
|---|---|---|
| in,out | font | The Font struct to which to add the table. |

Definition at line 2269 of file hex2otf.c.

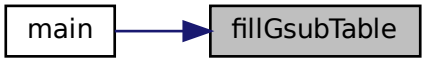
```
2270 {
2271     Buffer *gsub = newBuffer (38);
2272     addTable (font, ”GSUB”, gsub);
```

```
2273     cacheU16 (gsub, 1); // majorVersion
2274     cacheU16 (gsub, 0); // minorVersion
2275     cacheU16 (gsub, 10); // scriptListOffset
2276     cacheU16 (gsub, 34); // featureListOffset
2277     cacheU16 (gsub, 36); // lookupListOffset
2278     { // ScriptList table
2279         cacheU16 (gsub, 2); // scriptCount
2280         { // scriptRecords[0]
2281             cacheBytes (gsub, "DFLT", 4); // scriptTag
2282             cacheU16 (gsub, 14); // scriptOffset
2283         }
2284         { // scriptRecords[1]
2285             cacheBytes (gsub, "thai", 4); // scriptTag
2286             cacheU16 (gsub, 14); // scriptOffset
2287         }
2288         { // Script table
2289             cacheU16 (gsub, 4); // defaultLangSysOffset
2290             cacheU16 (gsub, 0); // langSysCount
2291             { // Default Language System table
2292                 cacheU16 (gsub, 0); // lookupOrderOffset
2293                 cacheU16 (gsub, 0); // requiredFeatureIndex
2294                 cacheU16 (gsub, 0); // featureIndexCount
2295             }
2296         }
2297     }
2298     { // Feature List table
2299         cacheU16 (gsub, 0); // featureCount
2300     }
2301     { // Lookup List Table
2302         cacheU16 (gsub, 0); // lookupCount
2303     }
2304 }
```

Here is the call graph for this function:




Here is the caller graph for this function:



#### 5.1.5.22 fillHeadTable()

```
void fillHeadTable (
                Font * font,
                enum LocaFormat locaFormat,
                pixels_t xMin )
```

Fill a ”head” font table.

The ”head” table contains font header information common to the whole font.

Parameters

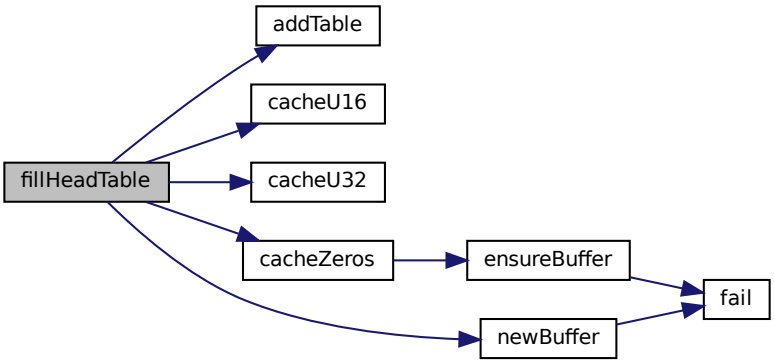
| | | |
|---|---|---|
| in,out | font | The Font struct to which to add the table. |
| in | locaFormat | The ”loca” offset index location table. |
| in | xMin | The minimum x-coordinate for a glyph. |

Definition at line 1853 of file hex2otf.c.

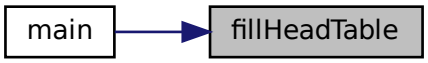
```
1854 {
1855     Buffer *head = newBuffer (56);
1856     addTable (font, "head", head);
1857     cacheU16 (head, 1); // majorVersion
1858     cacheU16 (head, 0); // minorVersion
1859     cacheZeros (head, 4); // fontRevision (unused)
1860     // The 'checksumAdjustment' field is a checksum of the entire file.
1861     // It is later calculated and written directly in the 'writeFont' function.
1862     cacheU32 (head, 0); // checksumAdjustment (placeholder)
1863     cacheU32 (head, 0x5f0f3cf5); // magicNumber
1864     const uint_fast16_t flags =
1865         + B1 ( 0) // baseline at y=0
1866         + B1 ( 1) // LSB at x=0 (doubtful; probably should be LSB=xMin)
1867         + B0 ( 2) // instructions may depend on point size
1868         + B0 ( 3) // force internal ppem to integers
1869         + B0 ( 4) // instructions may alter advance width
1870         + B0 ( 5) // not used in OpenType
1871         + B0 ( 6) // not used in OpenType
1872         + B0 ( 7) // not used in OpenType
1873         + B0 ( 8) // not used in OpenType
1874         + B0 ( 9) // not used in OpenType
1875         + B0 (10) // not used in OpenType
1876         + B0 (11) // font transformed
1877         + B0 (12) // font converted
1878         + B0 (13) // font optimized for ClearType
1879         + B0 (14) // last resort font
1880         + B0 (15) // reserved
1881     ;
1882     cacheU16 (head, flags); // flags
1883     cacheU16 (head, FUPEM); // unitsPerEm
1884     cacheZeros (head, 8); // created (unused)
1885     cacheZeros (head, 8); // modified (unused)
1886     cacheU16 (head, FU (xMin)); // xMin
1887     cacheU16 (head, FU (-DESCENDER)); // yMin
1888     cacheU16 (head, FU (font->maxWidth)); // xMax
1889     cacheU16 (head, FU (ASCENDER)); // yMax
1890     // macStyle (must agree with 'fsSelection' in 'OS/2' table)
1891     const uint_fast16_t macStyle =
1892         + B0 (0) // bold
1893         + B0 (1) // italic
1894         + B0 (2) // underline
1895         + B0 (3) // outline
1896         + B0 (4) // shadow
1897         + B0 (5) // condensed
1898         + B0 (6) // extended
1899         //    7-15  reserved
1900     ;
1901     cacheU16 (head, macStyle);
1902     cacheU16 (head, GLYPH_HEIGHT); // lowestRecPPEM
1903     cacheU16 (head, 2); // fontDirectionHint
1904     cacheU16 (head, locaFormat); // indexToLocFormat
```

```
1905     cacheU16 (head, 0); // glyphDataFormat
1906 }
```

Here is the call graph for this function:




Here is the caller graph for this function:




#### 5.1.5.23 fillHheaTable()

```
void fillHheaTable (
                Font * font,
                pixels_t xMin )
```

Fill a ”hhea” font table.

The ”hhea” table contains horizontal header information, for example left and right side bearings.

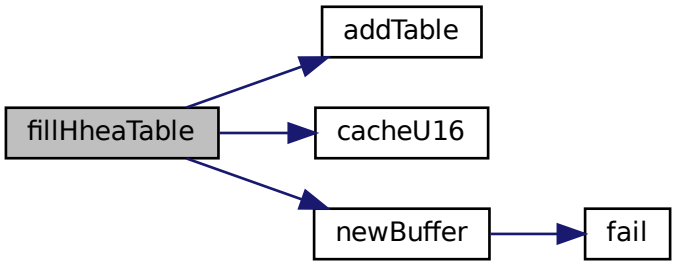
Parameters

| | | |
|---|---|---|
| in,out | font | The Font struct to which to add the table. |
| in | xMin | The minimum x-coordinate for a glyph. |

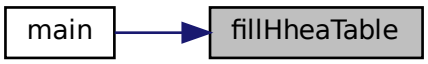
Definition at line 1918 of file hex2otf.c.

```
1919 {
1920     Buffer *hhea = newBuffer (36);
1921     addTable (font, "hhea", hhea);
1922     cacheU16 (hhea, 1); // majorVersion
1923     cacheU16 (hhea, 0); // minorVersion
1924     cacheU16 (hhea, FU (ASCENDER)); // ascender
1925     cacheU16 (hhea, FU (-DESCENDER)); // descender
1926     cacheU16 (hhea, FU (0)); // lineGap
1927     cacheU16 (hhea, FU (font->maxWidth)); // advanceWidthMax
1928     cacheU16 (hhea, FU (xMin)); // minLeftSideBearing
1929     cacheU16 (hhea, FU (0)); // minRightSideBearing (unused)
1930     cacheU16 (hhea, FU (font->maxWidth)); // xMaxExtent
1931     cacheU16 (hhea, 1); // caretSlopeRise
1932     cacheU16 (hhea, 0); // caretSlopeRun
1933     cacheU16 (hhea, 0); // caretOffset
1934     cacheU16 (hhea, 0); // reserved
1935     cacheU16 (hhea, 0); // reserved
1936     cacheU16 (hhea, 0); // reserved
1937     cacheU16 (hhea, 0); // reserved
1938     cacheU16 (hhea, 0); // metricDataFormat
1939     cacheU16 (hhea, font->glyphCount); // numberOfHMetrics
1940 }
```

Here is the call graph for this function:




Here is the caller graph for this function:



#### 5.1.5.24 fillHmtxTable()

```
void fillHmtxTable (
                Font * font )
```

Fill an ”hmtx” font table.

The ”hmtx” table contains horizontal metrics information.

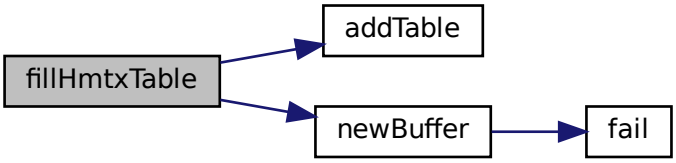
Parameters

| | | |
|---|---|---|
| in,out | font | The Font struct to which to add the table. |

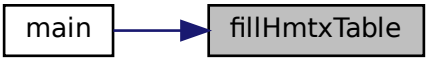
Definition at line 2087 of file hex2otf.c.

```
2088 {
2089     Buffer *hmtx = newBuffer (4 * font->glyphCount);
2090     addTable (font, ”hmtx”, hmtx);
2091     const Glyph *const glyphs = getBufferHead (font->glyphs);
2092     const Glyph *const glyphsEnd = getBufferTail (font->glyphs);
2093     for (const Glyph *glyph = glyphs; glyph < glyphsEnd; glyph++)
2094     {
2095         int_fast16_t aw = glyph->combining ? 0 : PW (glyph->byteCount);
2096         cacheU16 (hmtx, FU (aw)); // advanceWidth
2097         cacheU16 (hmtx, FU (glyph->lsb)); // lsb
2098     }
2099 }
```

Here is the call graph for this function:




Here is the caller graph for this function:



#### 5.1.5.25 fillMaxpTable()

```
void fillMaxpTable (
                Font * font,
                bool isCFF,
                uint_fast16_t maxPoints,
                uint_fast16_t maxContours )
```

Fill a ”maxp” font table.

The ”maxp” table contains maximum profile information, such as the memory required to contain the font.

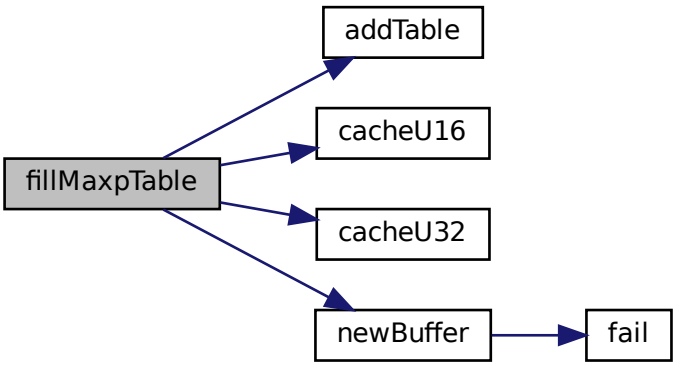
Parameters

| | | |
|---|---|---|
| in,out | font | The Font struct to which to add the table. |
| in | isCFF | true if a CFF font is included, false otherwise. |
| in | maxPoints | Maximum points in a non-composite glyph. |
| in | maxContours | Maximum contours in a non-composite glyph. |

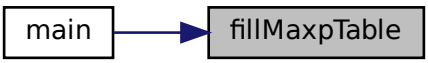
Definition at line 1954 of file hex2otf.c.

```
1956 {
1957     Buffer *maxp = newBuffer (32);
1958     addTable (font, "maxp", maxp);
1959     cacheU32 (maxp, isCFF ? 0x00005000 : 0x00010000); // version
1960     cacheU16 (maxp, font->glyphCount); // numGlyphs
1961     if (isCFF)
1962         return;
1963     cacheU16 (maxp, maxPoints); // maxPoints
1964     cacheU16 (maxp, maxContours); // maxContours
1965     cacheU16 (maxp, 0); // maxCompositePoints
1966     cacheU16 (maxp, 0); // maxCompositeContours
1967     cacheU16 (maxp, 0); // maxZones
1968     cacheU16 (maxp, 0); // maxTwilightPoints
1969     cacheU16 (maxp, 0); // maxStorage
1970     cacheU16 (maxp, 0); // maxFunctionDefs
1971     cacheU16 (maxp, 0); // maxInstructionDefs
1972     cacheU16 (maxp, 0); // maxStackElements
1973     cacheU16 (maxp, 0); // maxSizeOfInstructions
1974     cacheU16 (maxp, 0); // maxComponentElements
1975     cacheU16 (maxp, 0); // maxComponentDepth
1976 }
```

Here is the call graph for this function:




Here is the caller graph for this function:




#### 5.1.5.26 fillNameTable()

```
void fillNameTable (
            Font * font,
            NameStrings nameStrings )
```

Fill a ”name” font table.

The ”name” table contains name information, for example for Name IDs.

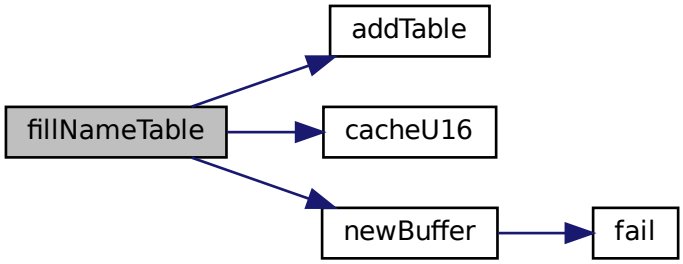
Parameters

| | | |
|---|---|---|
| in,out | font | The Font struct to which to add the table. |
| in | names | List of NameStrings. |

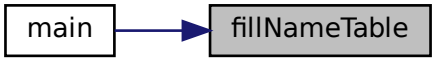
Definition at line 2366 of file hex2otf.c.

```
2367 {
2368     Buffer *name = newBuffer (2048);
2369     addTable (font, "name", name);
2370     size_t nameStringCount = 0;
2371     for (size_t i = 0; i < MAX_NAME_IDS; i++)
2372         nameStringCount += !!nameStrings[i];
2373     cacheU16 (name, 0); // version
2374     cacheU16 (name, nameStringCount); // count
2375     cacheU16 (name, 2 * 3 + 12 * nameStringCount); // storageOffset
2376     Buffer *stringData = newBuffer (1024);
2377     // nameRecord[]
2378     for (size_t i = 0; i < MAX_NAME_IDS; i++)
2379     {
2380         if (!nameStrings[i])
2381             continue;
2382         size_t offset = countBufferedBytes (stringData);
2383         cacheStringAsUTF16BE (stringData, nameStrings[i]);
2384         size_t length = countBufferedBytes (stringData) - offset;
2385         if (offset > U16MAX || length > U16MAX)
2386             fail ("Name strings are too long.");
2387         // Platform ID 0 (Unicode) is not well supported.
2388         // ID 3 (Windows) seems to be the best for compatibility.
2389         cacheU16 (name, 3); // platformID = Windows
2390         cacheU16 (name, 1); // encodingID = Unicode BMP
2391         cacheU16 (name, 0x0409); // languageID = en-US
2392         cacheU16 (name, i); // nameID
2393         cacheU16 (name, length); // length
2394         cacheU16 (name, offset); // stringOffset
2395     }
2396     cacheBuffer (name, stringData);
2397     freeBuffer (stringData);
2398 }
```

Here is the call graph for this function:




Here is the caller graph for this function:



#### 5.1.5.27 fillOS2Table()

void fillOS2Table (
Font * font )

Fill an "OS/2" font table.

The "OS/2" table contains OS/2 and Windows font metrics information.

Parameters

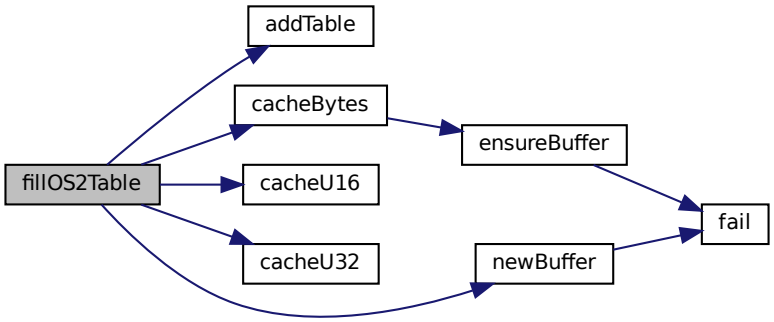
| | | |
|---|---|---|
| in,out | font | The Font struct to which to add the table. |

Definition at line 1986 of file hex2otf.c.

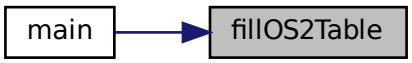
```
1987 {
1988     Buffer *os2 = newBuffer (100);
1989     addTable (font, "OS/2", os2);
1990     cacheU16 (os2, 5); // version
1991     // HACK: Average glyph width is not actually calculated.
1992     cacheU16 (os2, FU (font->maxWidth)); // xAvgCharWidth
1993     cacheU16 (os2, 400); // usWeightClass = Normal
1994     cacheU16 (os2, 5); // usWidthClass = Medium
1995     const uint_fast16_t typeFlags =
1996         + B0 (0) // reserved
1997         // usage permissions, one of:
1998             // Default: Installable embedding
1999             + B0 (1) // Restricted License embedding
2000             + B0 (2) // Preview & Print embedding
2001             + B0 (3) // Editable embedding
2002         //    4-7   reserved
2003         + B0 (8) // no subsetting
2004         + B0 (9) // bitmap embedding only
2005         //    10-15 reserved
2006     ;
2007     cacheU16 (os2, typeFlags); // fsType
2008     cacheU16 (os2, FU (5)); // ySubscriptXSize
2009     cacheU16 (os2, FU (7)); // ySubscriptYSize
2010     cacheU16 (os2, FU (0)); // ySubscriptXOffset
2011     cacheU16 (os2, FU (1)); // ySubscriptYOffset
2012     cacheU16 (os2, FU (5)); // ySuperscriptXSize
2013     cacheU16 (os2, FU (7)); // ySuperscriptYSize
2014     cacheU16 (os2, FU (0)); // ySuperscriptXOffset
2015     cacheU16 (os2, FU (4)); // ySuperscriptYOffset
2016     cacheU16 (os2, FU (1)); // yStrikeoutSize
2017     cacheU16 (os2, FU (5)); // yStrikeoutPosition
2018     cacheU16 (os2, 0x080a); // sFamilyClass = Sans Serif, Matrix
2019     const byte panose[] =
2020     {
2021         2, // Family Kind = Latin Text
2022         11, // Serif Style = Normal Sans
2023         4, // Weight = Thin
2024         // Windows would render all glyphs to the same width,
2025         // if 'Proportion' is set to 'Monospaced' (as Unifont should be).
2026         // 'Condensed' is the best alternative according to metrics.
2027         6, // Proportion = Condensed
2028         2, // Contrast = None
2029         2, // Stroke = No Variation
2030         2, // Arm Style = Straight Arms
2031         8, // Letterform = Normal/Square
2032         2, // Midline = Standard/Trimmed
2033         4, // X-height = Constant/Large
2034     };
2035     cacheBytes (os2, panose, sizeof panose); // panose
2036     // HACK: All defined Unicode ranges are marked functional for convenience.
2037     cacheU32 (os2, 0xffffffff); // ulUnicodeRange1
2038     cacheU32 (os2, 0xffffffff); // ulUnicodeRange2
2039     cacheU32 (os2, 0xffffffff); // ulUnicodeRange3
```

```
2040     cacheU32 (os2, 0x0effffff); // ulUnicodeRange4
2041     cacheBytes (os2, "GNU ", 4); // achVendID
2042     // fsSelection (must agree with 'macStyle' in 'head' table)
2043     const uint_fast16_t selection =
2044         + B0 (0) // italic
2045         + B0 (1) // underscored
2046         + B0 (2) // negative
2047         + B0 (3) // outlined
2048         + B0 (4) // strikeout
2049         + B0 (5) // bold
2050         + B1 (6) // regular
2051         + B1 (7) // use sTypo* metrics in this table
2052         + B1 (8) // font name conforms to WWS model
2053         + B0 (9) // oblique
2054         //    10-15 reserved
2055     ;
2056     cacheU16 (os2, selection);
2057     const Glyph *glyphs = getBufferHead (font->glyphs);
2058     uint_fast32_t first = glyphs[1].codePoint;
2059     uint_fast32_t last = glyphs[font->glyphCount - 1].codePoint;
2060     cacheU16 (os2, first < U16MAX ? first : U16MAX); // usFirstCharIndex
2061     cacheU16 (os2, last  < U16MAX ? last  : U16MAX); // usLastCharIndex
2062     cacheU16 (os2, FU (ASCENDER)); // sTypoAscender
2063     cacheU16 (os2, FU (-DESCENDER)); // sTypoDescender
2064     cacheU16 (os2, FU (0)); // sTypoLineGap
2065     cacheU16 (os2, FU (ASCENDER)); // usWinAscent
2066     cacheU16 (os2, FU (DESCENDER)); // usWinDescent
2067     // HACK: All reasonable code pages are marked functional for convenience.
2068     cacheU32 (os2, 0x603f01ff); // ulCodePageRange1
2069     cacheU32 (os2, 0xffff0000); // ulCodePageRange2
2070     cacheU16 (os2, FU (8)); // sxHeight
2071     cacheU16 (os2, FU (10)); // sCapHeight
2072     cacheU16 (os2, 0); // usDefaultChar
2073     cacheU16 (os2, 0x20); // usBreakChar
2074     cacheU16 (os2, 0); // usMaxContext
2075     cacheU16 (os2, 0); // usLowerOpticalPointSize
2076     cacheU16 (os2, 0xffff); // usUpperOpticalPointSize
2077 }
```

Here is the call graph for this function:



Here is the caller graph for this function:




#### 5.1.5.28 fillPostTable()

```
void fillPostTable (
                Font * font )
```

Fill a ”post” font table.

The ”post” table contains information for PostScript printers.

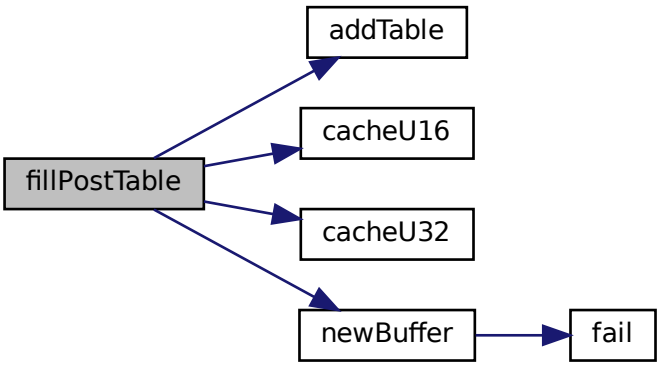
Parameters

| | | |
|---|---|---|
| in,out | font | The Font struct to which to add the table. |

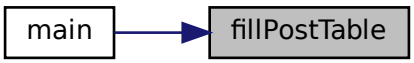
Definition at line 2218 of file hex2otf.c.

```
2219 {
2220     Buffer *post = newBuffer (32);
2221     addTable (font, ”post”, post);
2222     cacheU32 (post, 0x00030000); // version = 3.0
2223     cacheU32 (post, 0); // italicAngle
2224     cacheU16 (post, 0); // underlinePosition
2225     cacheU16 (post, 1); // underlineThickness
2226     cacheU32 (post, 1); // isFixedPitch
2227     cacheU32 (post, 0); // minMemType42
2228     cacheU32 (post, 0); // maxMemType42
2229     cacheU32 (post, 0); // minMemType1
2230     cacheU32 (post, 0); // maxMemType1
2231 }
```

Here is the call graph for this function:




Here is the caller graph for this function:




#### 5.1.5.29 fillTrueType()

```
void fillTrueType (
            Font * font,
            enum LocaFormat * format,
            uint_fast16_t * maxPoints,
            uint_fast16_t * maxContours )
```

Add a TrueType table to a font.

Parameters

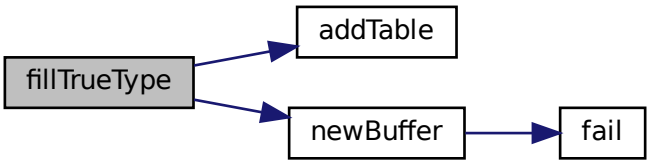
| | | |
|---|---|---|
| in,out | font | Pointer to a Font struct to contain the TrueType table. |
| in | format | The TrueType ”loca” table format, Offset16 or Offset32. |
| in | names | List of NameStrings. |

Definition at line 1597 of file hex2otf.c.

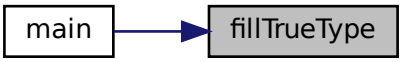
```
1599 {
1600     Buffer *glyf = newBuffer (65536);
1601     addTable (font, "glyf", glyf);
1602     Buffer *loca = newBuffer (4 * (font->glyphCount + 1));
1603     addTable (font, "loca", loca);
1604     *format = LOCA_OFFSET32;
1605     Buffer *endPoints = newBuffer (256);
1606     Buffer *flags = newBuffer (256);
1607     Buffer *xs = newBuffer (256);
1608     Buffer *ys = newBuffer (256);
1609     Buffer *outline = newBuffer (1024);
1610     Glyph *const glyphs = getBufferHead (font->glyphs);
1611     const Glyph *const glyphsEnd = getBufferTail (font->glyphs);
1612     for (Glyph *glyph = glyphs; glyph < glyphsEnd; glyph++)
1613     {
1614         cacheU32 (loca, countBufferedBytes (glyf));
1615         pixels_t rx = -glyph->pos;
1616         pixels_t ry = DESCENDER;
1617         pixels_t xMin = GLYPH_MAX_WIDTH, xMax = 0;
1618         pixels_t yMin = ASCENDER, yMax = -DESCENDER;
1619         resetBuffer (endPoints);
1620         resetBuffer (flags);
1621         resetBuffer (xs);
1622         resetBuffer (ys);
1623         resetBuffer (outline);
1624         buildOutline (outline, glyph->bitmap, glyph->byteCount, FILL_RIGHT);
1625         uint_fast32_t pointCount = 0, contourCount = 0;
1626         for (const pixels_t *p = getBufferHead (outline),
1627             *const end = getBufferTail (outline); p < end;)
1628         {
1629             const enum ContourOp op = *p++;
1630             if (op == OP_CLOSE)
1631             {
1632                 contourCount++;
1633                 assert (contourCount <= U16MAX);
1634                 cacheU16 (endPoints, pointCount - 1);
1635                 continue;
1636             }
1637             assert (op == OP_POINT);
1638             pointCount++;
1639             assert (pointCount <= U16MAX);
1640             const pixels_t x = *p++, y = *p++;
1641             uint_fast8_t pointFlags =
1642                 + B1 (0) // point is on curve
1643                 + BX (1, x != rx) // x coordinate is 1 byte instead of 2
1644                 + BX (2, y != ry) // y coordinate is 1 byte instead of 2
1645                 + B0 (3) // repeat
1646                 + BX (4, x >= rx) // when x is 1 byte: x is positive;
1647                                   // when x is 2 bytes: x unchanged and omitted
1648                 + BX (5, y >= ry) // when y is 1 byte: y is positive;
1649                                   // when y is 2 bytes: y unchanged and omitted
1650                 + B1 (6) // contours may overlap
1651                 + B0 (7) // reserved
1652             ;
1653             cacheU8 (flags, pointFlags);
1654             if (x != rx)
1655                 cacheU8 (xs, FU (x > rx ? x - rx : rx - x));
1656             if (y != ry)
1657                 cacheU8 (ys, FU (y > ry ? y - ry : ry - y));
1658             if (x < xMin) xMin = x;
1659             if (y < yMin) yMin = y;
1660             if (x > xMax) xMax = x;
1661             if (y > yMax) yMax = y;
1662             rx = x;
1663             ry = y;
1664         }
1665         if (contourCount == 0)
1666             continue; // blank glyph is indicated by the 'loca' table
1667         glyph->lsb = glyph->pos + xMin;
1668         cacheU16 (glyf, contourCount); // numberOfContours
1669         cacheU16 (glyf, FU (glyph->pos + xMin)); // xMin
1670         cacheU16 (glyf, FU (yMin)); // yMin
1671         cacheU16 (glyf, FU (glyph->pos + xMax)); // xMax
1672         cacheU16 (glyf, FU (yMax)); // yMax
1673         cacheBuffer (glyf, endPoints); // endPtsOfContours[]
1674         cacheU16 (glyf, 0); // instructionLength
1675         cacheBuffer (glyf, flags); // flags[]
1676         cacheBuffer (glyf, xs); // xCoordinates[]
1677         cacheBuffer (glyf, ys); // yCoordinates[]
1678         if (pointCount > *maxPoints)
```

```
1679             *maxPoints = pointCount;
1680         if (contourCount > *maxContours)
1681             *maxContours = contourCount;
1682     }
1683     cacheU32 (loca, countBufferedBytes (glyf));
1684     freeBuffer (endPoints);
1685     freeBuffer (flags);
1686     freeBuffer (xs);
1687     freeBuffer (ys);
1688     freeBuffer (outline);
1689 }
```

Here is the call graph for this function:




Here is the caller graph for this function:




#### 5.1.5.30 freeBuffer()

void freeBuffer (
Buffer * buf )

Free the memory previously allocated for a buffer.

This function frees the memory allocated to an array of type Buffer *.

Parameters

| | | |
|---|---|---|
| in | buf | The pointer to an array of type Buffer *. |

Definition at line 337 of file hex2otf.c.

```
338 {
339     free (buf->begin);
340     buf->capacity = 0;
341 }
```

#### 5.1.5.31 initBuffers()

```
void initBuffers (
                size_t count )
```

Initialize an array of buffer pointers to all zeroes.

This function initializes the "allBuffers" array of buffer pointers to all zeroes.

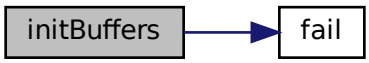
Parameters

| | | |
|---|---|---|
| in | count | The number of buffer array pointers to allocate. |

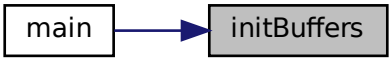
Definition at line 152 of file hex2otf.c.

```
153 {
154     assert (count > 0);
155     assert (bufferCount == 0); // uninitialized
156     allBuffers = calloc (count, sizeof *allBuffers);
157     if (!allBuffers)
158         fail ("Failed to initialize buffers.");
159     bufferCount = count;
160     nextBufferIndex = 0;
161 }
```

Here is the call graph for this function:




Here is the caller graph for this function:



#### 5.1.5.32 main()

```
int main (
            int argc,
            char * argv[] )
```

The main function.

Parameters

| | | |
|---|---|---|
| in | argc | The number of command-line arguments. |
| in | argv | The array of command-line arguments. |

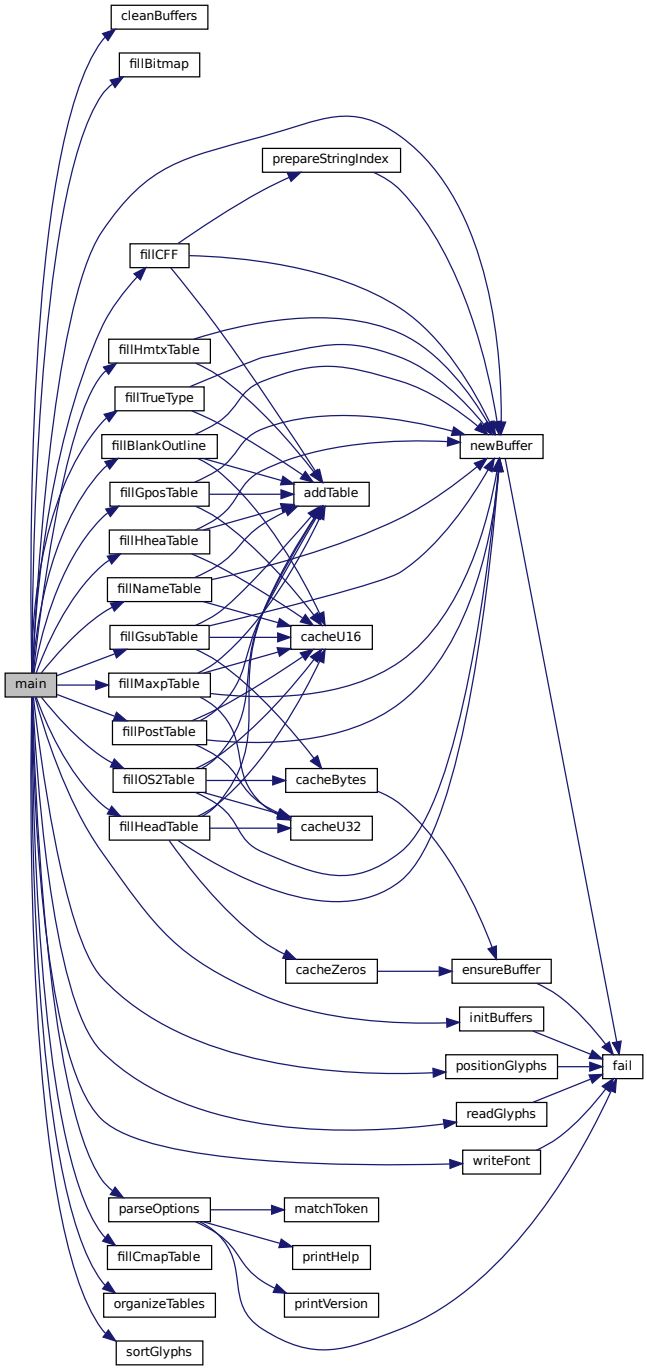
Returns

EXIT_FAILURE upon fatal error, EXIT_SUCCESS otherwise.

Definition at line 2603 of file hex2otf.c.

```
2604 {
2605     initBuffers (16);
2606     atexit (cleanBuffers);
2607     Options opt = parseOptions (argv);
2608     Font font;
2609     font.tables = newBuffer (sizeof (Table) * 16);
2610     font.glyphs = newBuffer (sizeof (Glyph) * MAX_GLYPHS);
2611     readGlyphs (&font, opt.hex);
2612     sortGlyphs (&font);
2613     enum LocaFormat loca = LOCA_OFFSET16;
2614     uint_fast16_t maxPoints = 0, maxContours = 0;
2615     pixels_t xMin = 0;
2616     if (opt.pos)
2617         positionGlyphs (&font, opt.pos, &xMin);
2618     if (opt.gpos)
2619         fillGposTable (&font);
2620     if (opt.gsub)
2621         fillGsubTable (&font);
2622     if (opt.cff)
2623         fillCFF (&font, opt.cff, opt.nameStrings);
2624     if (opt.truetype)
2625         fillTrueType (&font, &loca, &maxPoints, &maxContours);
2626     if (opt.blankOutline)
2627         fillBlankOutline (&font);
2628     if (opt.bitmap)
2629         fillBitmap (&font);
2630     fillHeadTable (&font, loca, xMin);
2631     fillHheaTable (&font, xMin);
2632     fillMaxpTable (&font, opt.cff, maxPoints, maxContours);
2633     fillOS2Table (&font);
2634     fillNameTable (&font, opt.nameStrings);
2635     fillHmtxTable (&font);
2636     fillCmapTable (&font);
2637     fillPostTable (&font);
2638     organizeTables (&font, opt.cff);
2639     writeFont (&font, opt.cff, opt.out);
2640     return EXIT_SUCCESS;
2641 }
```

Here is the call graph for this function:



#### 5.1.5.33 matchToken()

```
const char* matchToken (
                const char * operand,
                const char * key,
                char delimiter )
```

Match a command line option with its key for enabling.

Parameters

| | | |
|---|---|---|
| in | operand | A pointer to the specified operand. |
| in | key | Pointer to the option structure. |
| in | delimeter | The delimiter to end searching. |

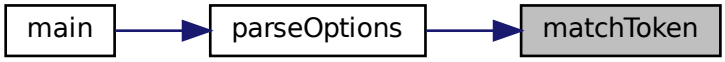
Returns

Pointer to the first character of the desired option.

Definition at line 2470 of file hex2otf.c.

```
2471 {
2472     while (*key)
2473         if (*operand++ != *key++)
2474             return NULL;
2475     if (!*operand || *operand++ == delimiter)
2476         return operand;
2477     return NULL;
2478 }
```

Here is the caller graph for this function:




#### 5.1.5.34 newBuffer()

```
Buffer* newBuffer (
                size_t initialCapacity )
```

Create a new buffer.

This function creates a new buffer array of type Buffer, with an initial size of initialCapacity elements.

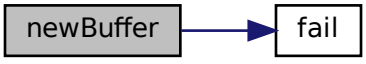
Parameters

| | | |
|---|---|---|
| in | initialCapacity | The initial number of elements in the buffer. |

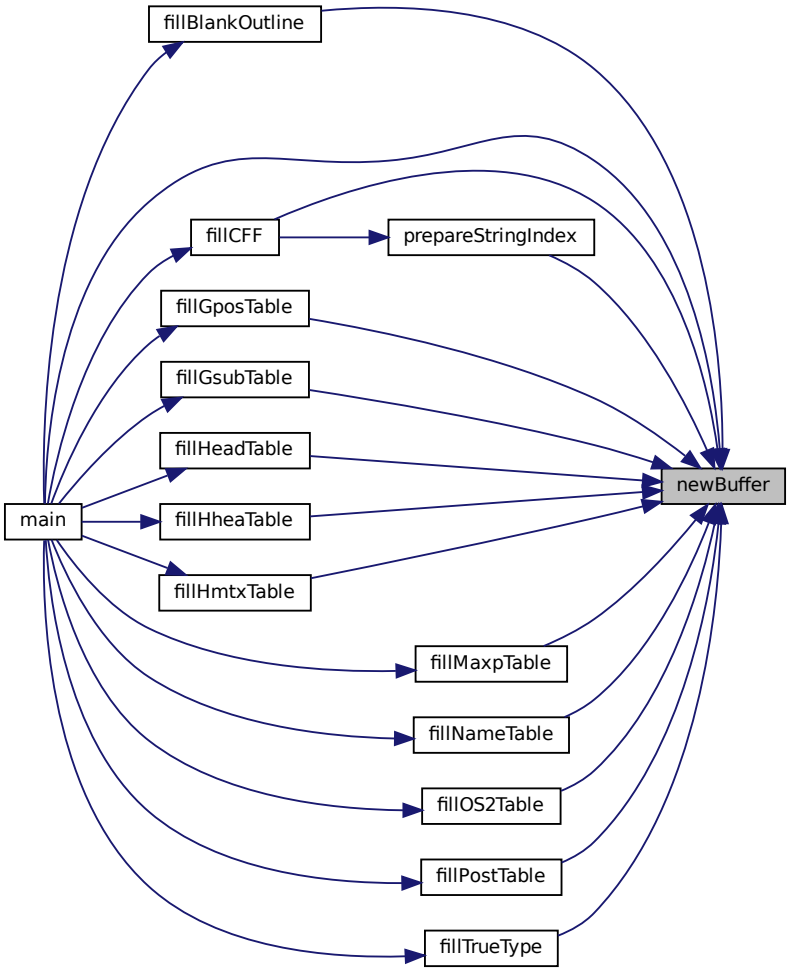
Definition at line 188 of file hex2otf.c.

```
189 {
190     assert (initialCapacity > 0);
191     Buffer *buf = NULL;
192     size_t sentinel = nextBufferIndex;
193     do
194     {
195         if (nextBufferIndex == bufferCount)
196             nextBufferIndex = 0;
197         if (allBuffers[nextBufferIndex].capacity == 0)
198         {
199             buf = &allBuffers[nextBufferIndex++];
200             break;
201         }
202     } while (++nextBufferIndex != sentinel);
203     if (!buf) // no existing buffer available
204     {
205         size_t newSize = sizeof (Buffer) * bufferCount * 2;
206         void *extended = realloc (allBuffers, newSize);
207         if (!extended)
208             fail ("Failed to create new buffers.");
209         allBuffers = extended;
210         memset (allBuffers + bufferCount, 0, sizeof (Buffer) * bufferCount);
211         buf = &allBuffers[bufferCount];
212         nextBufferIndex = bufferCount + 1;
213         bufferCount *= 2;
214     }
215     buf->begin = malloc (initialCapacity);
216     if (!buf->begin)
217         fail ("Failed to allocate %zu bytes of memory.", initialCapacity);
218     buf->capacity = initialCapacity;
219     buf->next = buf->begin;
220     buf->end = buf->begin + initialCapacity;
221     return buf;
222 }
```

Here is the call graph for this function:



Here is the caller graph for this function:




#### 5.1.5.35 organizeTables()

```
void organizeTables (
                Font * font,
                bool isCFF )
```

Sort tables according to OpenType recommendations.

The various tables in a font are sorted in an order recommended for TrueType font files.

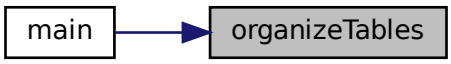
Parameters

| | | |
|---|---|---|
| in,out | font | The font in which to sort tables. |
| in | isCFF | True iff Compact Font Format (CFF) is being used. |

Definition at line 711 of file hex2otf.c.

```
712 {
713     const char *const cffOrder[] = {"head","hhea","maxp","OS/2","name",
714         "cmap","post","CFF ",NULL};
715     const char *const truetypeOrder[] = {"head","hhea","maxp","OS/2",
716         "hmtx","LTSH","VDMX","hdmx","cmap","fpgm","prep","cvt ","loca",
717         "glyf","kern","name","post","gasp","PCLT","DSIG",NULL};
718     const char *const *const order = isCFF ? cffOrder : truetypeOrder;
719     Table *unordered = getBufferHead (font->tables);
720     const Table *const tablesEnd = getBufferTail (font->tables);
721     for (const char *const *p = order; *p; p++)
722     {
723         uint_fast32_t tag = tagAsU32 (*p);
724         for (Table *t = unordered; t < tablesEnd; t++)
725         {
726             if (t->tag != tag)
727                 continue;
728             if (t != unordered)
729             {
730                 Table temp = *unordered;
731                 *unordered = *t;
732                 *t = temp;
733             }
734             unordered++;
735             break;
736         }
737     }
738 }
```

Here is the caller graph for this function:




### 5.1.5.36 parseOptions()

```
Options parseOptions (
                char *const argv[const ] )
```

Parse command line options.

| Option | Data Type | Description |
|---|---|---|
| truetype | bool | Generate TrueType outlines |
| blankOutline | bool | Generate blank outlines |
| bitmap | bool | Generate embedded bitmap |
| gpos | bool | Generate a dummy GPOS table |
| gsub | bool | Generate a dummy GSUB table |
| cff | int | Generate CFF 1 or CFF 2 outlines |
| hex | const char * | Name of Unifont .hex file |
| pos | const char * | Name of Unifont combining data file |
| out | const char * | Name of output font file |
| nameStrings | NameStrings | Array of TrueType font Name IDs |

### Parameters

| | | |
|---|---|---|
| in | argv | Pointer to array of command line options. |

### Returns

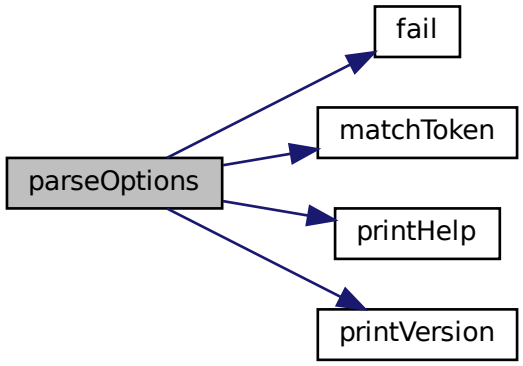
Data structure to hold requested command line options.

Definition at line 2500 of file hex2otf.c.

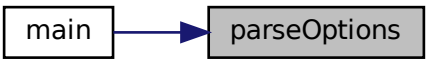
```
2501 {
2502     Options opt = {0}; // all options default to 0, false and NULL
2503     const char *format = NULL;
2504     struct StringArg
2505     {
2506         const char *const key;
2507         const char **const value;
2508     } strArgs[] =
2509     {
2510         {"hex", &opt.hex},
2511         {"pos", &opt.pos},
2512         {"out", &opt.out},
2513         {"format", &format},
2514         {NULL, NULL} // sentinel
2515     };
2516     for (char *const *argp = argv + 1; *argp; argp++)
2517     {
2518         const char *const arg = *argp;
2519         struct StringArg *p;
2520         const char *value = NULL;
2521         if (strcmp (arg, "--help") == 0)
2522             printHelp ();
2523         if (strcmp (arg, "--version") == 0)
2524             printVersion ();
2525         for (p = strArgs; p->key; p++)
2526             if ((value = matchToken (arg, p->key, '=')))
2527                 break;
2528         if (p->key)
2529         {
2530             if (!*value)
2531                 fail ("Empty argument: '%s'.", p->key);
2532             if (*p->value)
2533                 fail ("Duplicate argument: '%s'.", p->key);
2534             *p->value = value;
2535         }
2536         else // shall be a name string
2537         {
2538             char *endptr;
2539             unsigned long id = strtoul (arg, &endptr, 10);
2540             if (endptr == arg || id >= MAX_NAME_IDS || *endptr != '=')
2541                 fail ("Invalid argument: '%s'.", arg);
2542             endptr++; // skip '='
2543             if (opt.nameStrings[id])
2544                 fail ("Duplicate name ID: %lu.", id);
2545             opt.nameStrings[id] = endptr;
2546         }
2547     }
2548     if (!opt.hex)
```

```
2549        fail ("Hex file is not specified.");
2550    if (opt.pos && opt.pos[0] == '\0')
2551        opt.pos = NULL; // Position file is optional. Empty path means none.
2552    if (!opt.out)
2553        fail ("Output file is not specified.");
2554    if (!format)
2555        fail ("Format is not specified.");
2556    for (const NamePair *p = defaultNames; p->str; p++)
2557        if (!opt.nameStrings[p->id])
2558            opt.nameStrings[p->id] = p->str;
2559    bool cff = false, cff2 = false;
2560    struct Symbol
2561    {
2562        const char *const key;
2563        bool *const found;
2564    } symbols[] =
2565    {
2566        {"cff", &cff},
2567        {"cff2", &cff2},
2568        {"truetype", &opt.truetype},
2569        {"blank", &opt.blankOutline},
2570        {"bitmap", &opt.bitmap},
2571        {"gpos", &opt.gpos},
2572        {"gsub", &opt.gsub},
2573        {NULL, NULL} // sentinel
2574    };
2575    while (*format)
2576    {
2577        const struct Symbol *p;
2578        const char *next = NULL;
2579        for (p = symbols; p->key; p++)
2580            if ((next = matchToken (format, p->key, ',')))
2581                break;
2582        if (!p->key)
2583            fail ("Invalid format.");
2584        *p->found = true;
2585        format = next;
2586    }
2587    if (cff + cff2 + opt.truetype + opt.blankOutline > 1)
2588        fail ("At most one outline format can be accepted.");
2589    if (!(cff || cff2 || opt.truetype || opt.bitmap))
2590        fail ("Invalid format.");
2591    opt.cff = cff + cff2 * 2;
2592    return opt;
2593 }
```

Here is the call graph for this function:



Here is the caller graph for this function:




#### 5.1.5.37 positionGlyphs()

```
void positionGlyphs (
                Font * font,
                const char * fileName,
                pixels_t * xMin )
```

Position a glyph within a 16-by-16 pixel bounding box.

Position a glyph within the 16-by-16 pixel drawing area and note whether or not the glyph is a combining character.

N.B.: Glyphs must be sorted by code point before calling this function.

Parameters

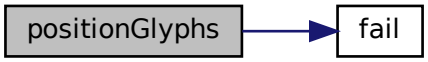
| | | |
|---|---|---|
| in,out | font | Font data structure pointer to store glyphs. |
| in | fileName | Name of glyph file to read. |
| in | xMin | Minimum x-axis value (for left side bearing). |

Definition at line 1061 of file hex2otf.c.

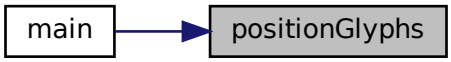
```
1062 {
1063     *xMin = 0;
1064     FILE *file = fopen (fileName, "r");
1065     if (!file)
1066         fail ("Failed to open file '%s'.", fileName);
1067     Glyph *glyphs = getBufferHead (font->glyphs);
1068     const Glyph *const endGlyph = glyphs + font->glyphCount;
1069     Glyph *nextGlyph = &glyphs[1]; // predict and avoid search
1070     for (;;)
1071     {
1072         uint_fast32_t codePoint;
1073         if (readCodePoint (&codePoint, fileName, file))
1074             break;
1075         Glyph *glyph = nextGlyph;
1076         if (glyph == endGlyph || glyph->codePoint != codePoint)
1077         {
1078             // Prediction failed. Search.
1079             const Glyph key = { .codePoint = codePoint };
1080             glyph = bsearch (&key, glyphs + 1, font->glyphCount - 1,
1081                 sizeof key, byCodePoint);
1082             if (!glyph)
```

```
1083                 fail ("Glyph "PRI_CP" is positioned but not defined.",
1084                     codePoint);
1085         }
1086         nextGlyph = glyph + 1;
1087         char s[8];
1088         if (!fgets (s, sizeof s, file))
1089             fail ("%s: Read error.", fileName);
1090         char *end;
1091         const long value = strtol (s, &end, 10);
1092         if (*end != '\n' && *end != '\0')
1093             fail ("Position of glyph "PRI_CP" is invalid.", codePoint);
1094         // Currently no glyph is moved to the right,
1095         // so positive position is considered out of range.
1096         // If this limit is to be lifted,
1097         // 'xMax' of bounding box in 'head' table shall also be updated.
1098         if (value < -GLYPH_MAX_WIDTH || value > 0)
1099             fail ("Position of glyph "PRI_CP" is out of range.", codePoint);
1100         glyph->combining = true;
1101         glyph->pos = value;
1102         glyph->lsb = value; // updated during outline generation
1103         if (value < *xMin)
1104             *xMin = value;
1105     }
1106     fclose (file);
1107 }
```

Here is the call graph for this function:




Here is the caller graph for this function:




#### 5.1.5.38 prepareOffsets()

```
void prepareOffsets (
                size_t * sizes )
```

Prepare 32-bit glyph offsets in a font table.

Parameters

| | | |
|---|---|---|
| in | sizes | Array of glyph sizes, for offset calculations. |

Definition at line 1275 of file hex2otf.c.

```
1276 {
1277     size_t *p = sizes;
1278     for (size_t *i = sizes + 1; *i; i++)
1279         *i += *p++;
1280     if (*p > 2147483647U) // offset not representable
1281         fail ("CFF table is too large.");
1282 }
```

Here is the call graph for this function:


### 5.1.5.39 prepareStringIndex()

```
Buffer* prepareStringIndex (
                const NameStrings names )
```

Prepare a font name string index.

Parameters

| | | |
|---|---|---|
| in | names | List of name strings. |

Returns

Pointer to a Buffer struct containing the string names.

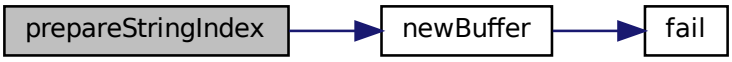
Get the number of elements in array char *strings[].

Definition at line 1291 of file hex2otf.c.

```
1292 {
1293     Buffer *buf = newBuffer (256);
1294     assert (names[6]);
1295     const char *strings[] = {"Adobe", "Identity", names[6]};
1296 /// Get the number of elements in array char *strings[].
1297 #define stringCount (sizeof strings / sizeof *strings)
1298     static_assert (stringCount <= U16MAX, "too many strings");
```

```
1299     size_t offset = 1;
1300     size_t lengths[stringCount];
1301     for (size_t i = 0; i < stringCount; i++)
1302     {
1303         assert (strings[i]);
1304         lengths[i] = strlen (strings[i]);
1305         offset += lengths[i];
1306     }
1307     int offsetSize = 1 + (offset > 0xff)
1308                        + (offset > 0xffff)
1309                        + (offset > 0xffffff);
1310     cacheU16 (buf, stringCount); // count
1311     cacheU8 (buf, offsetSize); // offSize
1312     cacheU (buf, offset = 1, offsetSize); // offset[0]
1313     for (size_t i = 0; i < stringCount; i++)
1314         cacheU (buf, offset += lengths[i], offsetSize); // offset[i + 1]
1315     for (size_t i = 0; i < stringCount; i++)
1316         cacheBytes (buf, strings[i], lengths[i]);
1317 #undef stringCount
1318     return buf;
1319 }
```

Here is the call graph for this function:




Here is the caller graph for this function:


#### 5.1.5.40 printHelp()

void printHelp ( )

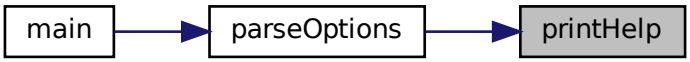
Print help message to stdout and then exit.

Print help message if invoked with the ”--help” option, and then exit successfully.

Definition at line 2426 of file hex2otf.c.

```
2426             {
2427     printf ("Synopsis: hex2otf <options>:\n\n");
2428     printf ("    hex=<filename>        Specify Unifont .hex input file.\n");
2429     printf ("    pos=<filename>        Specify combining file. (Optional)\n");
2430     printf ("    out=<filename>        Specify output font file.\n");
2431     printf ("    format=<f1>,<f2>,...  Specify font format(s); values:\n");
2432     printf ("                          cff\n");
2433     printf ("                          cff2\n");
2434     printf ("                          truetype\n");
2435     printf ("                          blank\n");
2436     printf ("                          bitmap\n");
2437     printf ("                          gpos\n");
2438     printf ("                          gsub\n");
2439     printf ("\nExample:\n\n");
2440     printf ("    hex2otf hex=Myfont.hex out=Myfont.otf format=cff\n\n");
2441     printf ("For more information, consult the hex2otf(1) man page.\n\n");
2442
2443     exit (EXIT_SUCCESS);
2444 }
```

Here is the caller graph for this function:




#### 5.1.5.41 printVersion()

void printVersion ( )

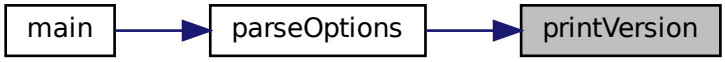
Print program version string on stdout.

Print program version if invoked with the "--version" option, and then exit successfully.

Definition at line 2407 of file hex2otf.c.

```
2407             {
2408     printf ("hex2otf (GNU Unifont) %s\n", VERSION);
2409     printf ("Copyright \u00A9 2022 \u4F55\u5FD7\u7FD4 (He Zhixiang)\n");
2410     printf ("License GPLv2+: GNU GPL version 2 or later\n");
2411     printf ("<https://gnu.org/licenses/gpl.html>\n");
2412     printf ("This is free software: you are free to change and\n");
2413     printf ("redistribute it.  There is NO WARRANTY, to the extent\n");
2414     printf ("permitted by law.\n");
2415
2416     exit (EXIT_SUCCESS);
2417 }
```

Here is the caller graph for this function:



#### 5.1.5.42 readCodePoint()

```
bool readCodePoint (
                uint_fast32_t * codePoint,
                const char * fileName,
                FILE * file )
```

Read up to 6 hexadecimal digits and a colon from file.

This function reads up to 6 hexadecimal digits followed by a colon from a file.

If the end of the file is reached, the function returns true. The file name is provided to include in an error message if the end of file was reached unexpectedly.

Parameters

| | | |
|---|---|---|
| out | codePoint | The Unicode code point. |
| in | fileName | The name of the input file. |
| in | file | Pointer to the input file stream. |

Returns

true if at end of file, false otherwise.

Definition at line 919 of file hex2otf.c.

```
920 {
921     *codePoint = 0;
922     uint_fast8_t digitCount = 0;
923     for (;;)
924     {
925         int c = getc (file);
926         if (isxdigit (c) && ++digitCount <= 6)
927         {
928             *codePoint = (*codePoint « 4) | nibbleValue (c);
929             continue;
930         }
931         if (c == ':' && digitCount > 0)
932             return false;
933         if (c == EOF)
934         {
935             if (digitCount == 0)
936                 return true;
937             if (feof (file))
938                 fail ("%s: Unexpected end of file.", fileName);
939             else
940                 fail ("%s: Read error.", fileName);
941         }
942         fail ("%s: Unexpected character: %#.2x.", fileName, (unsigned)c);
943     }
944 }
```

#### 5.1.5.43 readGlyphs()

```
void readGlyphs (
                Font * font,
                const char * fileName )
```

Read glyph definitions from a Unifont .hex format file.

This function reads in the glyph bitmaps contained in a Unifont .hex format file. These input files contain one glyph bitmap per line. Each line is of the form

<hexadecimal code point> ':' <hexadecimal bitmap sequence>

The code point field typically consists of 4 hexadecimal digits for a code point in Unicode Plane 0, and 6 hexadecimal digits for code points above Plane 0. The hexadecimal bitmap sequence is 32 hexadecimal digits long for a glyph that is 8 pixels wide by 16 pixels high, and 64 hexadecimal digits long for a glyph that is 16 pixels wide by 16 pixels high.

Parameters

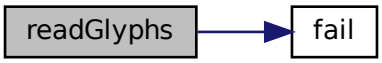
| | | |
|---|---|---|
| in,out | font | The font data structure to update with new glyphs. |
| in | fileName | The name of the Unifont .hex format input file. |

Definition at line 966 of file hex2otf.c.

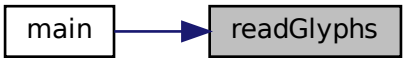
```
967 {
968     FILE *file = fopen (fileName, "r");
969     if (!file)
970         fail ("Failed to open file '%s'.", fileName);
971     uint_fast32_t glyphCount = 1; // for glyph 0
972     uint_fast8_t maxByteCount = 0;
973     { // Hard code the .notdef glyph.
974         const byte bitmap[] = "\0\0\0~fZZzvv~vv~\0\0"; // same as U+FFFD
975         const size_t byteCount = sizeof bitmap - 1;
976         assert (byteCount <= GLYPH_MAX_BYTE_COUNT);
977         assert (byteCount % GLYPH_HEIGHT == 0);
978         Glyph *notdef = getBufferSlot (font->glyphs, sizeof (Glyph));
979         memcpy (notdef->bitmap, bitmap, byteCount);
980         notdef->byteCount = maxByteCount = byteCount;
981         notdef->combining = false;
982         notdef->pos = 0;
983         notdef->lsb = 0;
984     }
985     for (;;)
986     {
987         uint_fast32_t codePoint;
988         if (readCodePoint (&codePoint, fileName, file))
989             break;
990         if (++glyphCount > MAX_GLYPHS)
991             fail ("OpenType does not support more than %lu glyphs.",
992                 MAX_GLYPHS);
993         Glyph *glyph = getBufferSlot (font->glyphs, sizeof (Glyph));
994         glyph->codePoint = codePoint;
995         glyph->byteCount = 0;
996         glyph->combining = false;
997         glyph->pos = 0;
998         glyph->lsb = 0;
999         for (byte *p = glyph->bitmap;; p++)
1000          {
1001              int h, l;
1002              if (isxdigit (h = getc (file)) && isxdigit (l = getc (file)))
1003              {
1004                  if (++glyph->byteCount > GLYPH_MAX_BYTE_COUNT)
1005                      fail ("Hex stream of "PRI_CP" is too long.", codePoint);
1006                  *p = nibbleValue (h) « 4 | nibbleValue (l);
1007              }
```

```
1008             else if (h == '\n' || (h == EOF && feof (file)))
1009                 break;
1010             else if (ferror (file))
1011                 fail ("%s: Read error.", fileName);
1012             else
1013                 fail ("Hex stream of "PRI_CP" is invalid.", codePoint);
1014         }
1015         if (glyph->byteCount % GLYPH_HEIGHT != 0)
1016             fail ("Hex length of "PRI_CP" is indivisible by glyph height %d.",
1017                 codePoint, GLYPH_HEIGHT);
1018         if (glyph->byteCount > maxByteCount)
1019             maxByteCount = glyph->byteCount;
1020     }
1021     if (glyphCount == 1)
1022         fail ("No glyph is specified.");
1023     font->glyphCount = glyphCount;
1024     font->maxWidth = PW (maxByteCount);
1025     fclose (file);
1026 }
```

Here is the call graph for this function:




Here is the caller graph for this function:




#### 5.1.5.44 sortGlyphs()

```
void sortGlyphs (
                Font * font )
```

Sort the glyphs in a font by Unicode code point.

This function reads in an array of glyphs and sorts them by Unicode code point. If a duplicate code point is encountered, that will result in a fatal error with an error message to stderr.

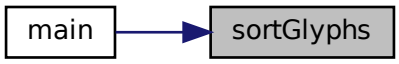
Parameters

| | | |
|---|---|---|
| in,out | font | Pointer to a Font structure with glyphs to sort. |

Definition at line 1119 of file hex2otf.c.

```
1120 {
1121     Glyph *glyphs = getBufferHead (font->glyphs);
1122     const Glyph *const glyphsEnd = getBufferTail (font->glyphs);
1123     glyphs++; // glyph 0 does not need sorting
1124     qsort (glyphs, glyphsEnd - glyphs, sizeof *glyphs, byCodePoint);
1125     for (const Glyph *glyph = glyphs; glyph < glyphsEnd - 1; glyph++)
1126     {
1127         if (glyph[0].codePoint == glyph[1].codePoint)
1128             fail ("Duplicate code point: "PRI_CP".", glyph[0].codePoint);
1129         assert (glyph[0].codePoint < glyph[1].codePoint);
1130     }
1131 }
```

Here is the caller graph for this function:




### 5.1.5.45 writeBytes()

```
void writeBytes (
                const byte bytes[],
                size_t count,
                FILE * file )
```

Write an array of bytes to an output file.

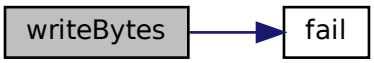
Parameters

| | | |
|---|---|---|
| in | bytes | An array of unsigned bytes to write. |
| in | file | The file pointer for writing, of type FILE *. |

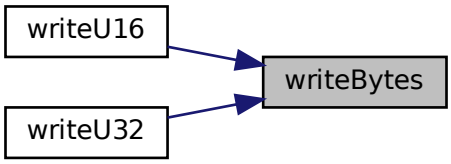
Definition at line 538 of file hex2otf.c.

```
539 {
540     if (fwrite (bytes, count, 1, file) != 1 && count != 0)
541         fail ("Failed to write %zu bytes to output file.", count);
542 }
```

Here is the call graph for this function:




Here is the caller graph for this function:




#### 5.1.5.46 writeFont()

```
void writeFont (
            Font * font,
            bool isCFF,
            const char * fileName )
```

Write OpenType font to output file.

This function writes the constructed OpenType font to the output file named ”filename”.

Parameters

| | | |
|---|---|---|
| in | font | Pointer to the font, of type Font *. |
| in | isCFF | Boolean indicating whether the font has CFF data. |
| in | filename | The name of the font file to create. |

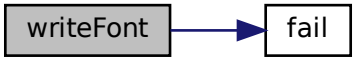
Add a byte shifted by 24, 16, 8, or 0 bits.

Definition at line 786 of file hex2otf.c.

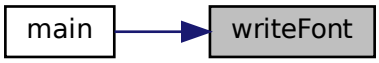
```
787 {
788     FILE *file = fopen (fileName, "wb");
789     if (!file)
790         fail ("Failed to open file '%s'.", fileName);
791     const Table *const tables = getBufferHead (font->tables);
792     const Table *const tablesEnd = getBufferTail (font->tables);
793     size_t tableCount = tablesEnd - tables;
794     assert (0 < tableCount && tableCount <= U16MAX);
795     size_t offset = 12 + 16 * tableCount;
796     uint_fast32_t totalChecksum = 0;
797     Buffer *tableRecords =
798         newBuffer (sizeof (struct TableRecord) * tableCount);
799     for (size_t i = 0; i < tableCount; i++)
800     {
801         struct TableRecord *record =
802             getBufferSlot (tableRecords, sizeof *record);
803         record->tag = tables[i].tag;
804         size_t length = countBufferedBytes (tables[i].content);
805 #if SIZE_MAX > U32MAX
806             if (offset > U32MAX)
807                 fail ("Table offset exceeded 4 GiB.");
808             if (length > U32MAX)
809                 fail ("Table size exceeded 4 GiB.");
810 #endif
811         record->length = length;
812         record->checksum = 0;
813         const byte *p = getBufferHead (tables[i].content);
814         const byte *const end = getBufferTail (tables[i].content);
815 
816 /// Add a byte shifted by 24, 16, 8, or 0 bits.
817 #define addByte(shift) \
818 if (p == end) \
819 break; \
820 record->checksum += (uint_fast32_t)*p++ « (shift);
821 
822         for (;;)
823         {
824             addByte (24)
825             addByte (16)
826             addByte (8)
827             addByte (0)
828         }
829         #undef addByte
830         cacheZeros (tables[i].content, (~length + 1U) & 3U);
831         record->offset = offset;
832         offset += countBufferedBytes (tables[i].content);
833         totalChecksum += record->checksum;
834     }
835     struct TableRecord *records = getBufferHead (tableRecords);
836     qsort (records, tableCount, sizeof *records, byTableTag);
837     // Offset Table
838     uint_fast32_t sfntVersion = isCFF ? 0x4f54544f : 0x00010000;
839     writeU32 (sfntVersion, file); // sfntVersion
840     totalChecksum += sfntVersion;
841     uint_fast16_t entrySelector = 0;
842     for (size_t k = tableCount; k != 1; k »= 1)
843         entrySelector++;
844     uint_fast16_t searchRange = 1 « (entrySelector + 4);
845     uint_fast16_t rangeShift = (tableCount - (1 « entrySelector)) « 4;
846     writeU16 (tableCount, file); // numTables
847     writeU16 (searchRange, file); // searchRange
848     writeU16 (entrySelector, file); // entrySelector
849     writeU16 (rangeShift, file); // rangeShift
850     totalChecksum += (uint_fast32_t)tableCount « 16;
851     totalChecksum += searchRange;
852     totalChecksum += (uint_fast32_t)entrySelector « 16;
853     totalChecksum += rangeShift;
854     // Table Records (always sorted by table tags)
855     for (size_t i = 0; i < tableCount; i++)
856     {
857         // Table Record
858         writeU32 (records[i].tag, file); // tableTag
859         writeU32 (records[i].checksum, file); // checkSum
860         writeU32 (records[i].offset, file); // offset
861         writeU32 (records[i].length, file); // length
862         totalChecksum += records[i].tag;
863         totalChecksum += records[i].checksum;
864         totalChecksum += records[i].offset;
865         totalChecksum += records[i].length;
866     }
```

```
867     freeBuffer (tableRecords);
868     for (const Table *table = tables; table < tablesEnd; table++)
869     {
870         if (table->tag == 0x68656164) // 'head' table
871         {
872             byte *begin = getBufferHead (table->content);
873             byte *end = getBufferTail (table->content);
874             writeBytes (begin, 8, file);
875             writeU32 (0xb1b0afbaU - totalChecksum, file); // checksumAdjustment
876             writeBytes (begin + 12, end - (begin + 12), file);
877             continue;
878         }
879         writeBuffer (table->content, file);
880     }
881     fclose (file);
882 }
```

Here is the call graph for this function:




Here is the caller graph for this function:




#### 5.1.5.47 writeU16()

void writeU16 (
uint_fast16_t value,
FILE * file )

Write an unsigned 16-bit value to an output file.

This function writes a 16-bit unsigned value in big-endian order to an output file specified with a file pointer.

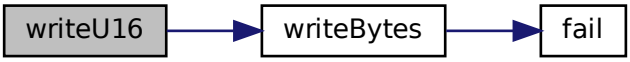
Parameters

| | | |
|---|---|---|
| in | value | The 16-bit value to write. |
| in | file | The file pointer for writing, of type FILE *. |

Definition at line 554 of file hex2otf.c.

```
555 {
556     byte bytes[] =
557     {
558         (value » 8) & 0xff,
559         (value     ) & 0xff,
560     };
561     writeBytes (bytes, sizeof bytes, file);
562 }
```

Here is the call graph for this function:




#### 5.1.5.48 writeU32()

```
void writeU32 (
                uint_fast32_t value,
                FILE * file )
```

Write an unsigned 32-bit value to an output file.

This function writes a 32-bit unsigned value in big-endian order to an output file specified with a file pointer.

Parameters

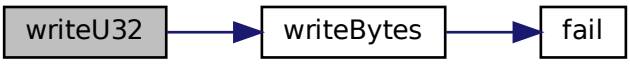
| | | |
|---|---|---|
| in | value | The 32-bit value to write. |
| in | file | The file pointer for writing, of type FILE *. |

Definition at line 574 of file hex2otf.c.

```
575 {
576     byte bytes[] =
577     {
578         (value » 24) & 0xff,
579         (value » 16) & 0xff,
580         (value »  8) & 0xff,
581         (value      ) & 0xff,
582     };
```

```
583     writeBytes (bytes, sizeof bytes, file);
584 }
```

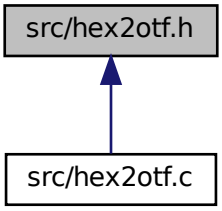
Here is the call graph for this function:




## 5.2 src/hex2otf.h File Reference

hex2otf.h - Header file for hex2otf.c

This graph shows which files directly or indirectly include this file:




### Data Structures

- struct NamePair

  Data structure for a font ID number and name character string.

### Macros

- #define UNIFONT_VERSION "15.0.05"

  Current Unifont version.
- #define DEFAULT_ID0 "Copyright © 1998-2022 Roman Czyborra, Paul Hardy, \Qianqian Fang, Andrew Miller, Johnnie Weaver, David Corbett, \Nils Moskopp, Rebecca Bettencourt, et al."
- #define DEFAULT_ID1 "Unifont"

Default NameID 1 string (Font Family)

- #define DEFAULT_ID2 "Regular"

  Default NameID 2 string (Font Subfamily)
- #define DEFAULT_ID5 "Version "UNIFONT_VERSION

  Default NameID 5 string (Version of the Name Table)
- #define DEFAULT_ID11 "https://unifoundry.com/unifont/"

  Default NameID 11 string (Font Vendor URL)
- #define DEFAULT_ID13 "Dual license: SIL Open Font License version 1.1, \and GNU GPL version 2 or later with the GNU Font Embedding Exception."

  Default NameID 13 string (License Description)
- #define DEFAULT_ID14 "http://unifoundry.com/LICENSE.txt, \https://scripts.sil.org/OFL"

  Default NameID 14 string (License Information URLs)
- #define NAMEPAIR(n) {(n), DEFAULT_ID##n}

  Macro to initialize name identifier codes to default values defined above.

## Typedefs

- typedef struct NamePair NamePair

  Data structure for a font ID number and name character string.

## Variables

- const NamePair defaultNames []

  Allocate array of NameID codes with default values.

### 5.2.1 Detailed Description

hex2otf.h - Header file for hex2otf.c

Copyright



Author

**何志翔** (He Zhixiang)

### 5.2.2 Macro Definition Documentation

#### 5.2.2.1 DEFAULT_ID0

#define DEFAULT_ID0 "Copyright © 1998-2022 Roman Czyborra, Paul Hardy, \Qianqian Fang, Andrew Miller, Johnnie Weaver, David Corbett, \Nils Moskopp, Rebecca Bettencourt, et al."

Define default strings for some TrueType font NameID strings.

| NameID | Description |
|---|---|
| 0 | Copyright Notice |
| 1 | Font Family |
| 2 | Font Subfamily |
| 5 | Version of the Name Table |
| 11 | URL of the Font Vendor |
| 13 | License Description |
| 14 | License Information URL |

Default NameID 0 string (Copyright Notice)

Definition at line 53 of file hex2otf.h.

### 5.2.3 Variable Documentation

#### 5.2.3.1 defaultNames

const NamePair defaultNames[]

Initial value:
=
{
  NAMEPAIR (0),
  NAMEPAIR (1),
  NAMEPAIR (2),
  NAMEPAIR (5),
  NAMEPAIR (11),
  NAMEPAIR (13),
  NAMEPAIR (14),
  {0, NULL}
}

Allocate array of NameID codes with default values.

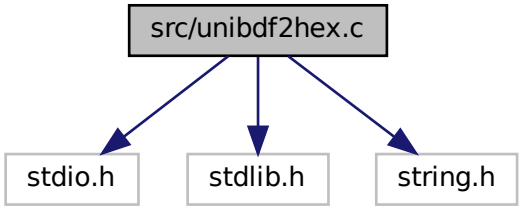
This array contains the default values for several TrueType NameID strings, as defined above in this file. Strings are assigned using the NAMEPAIR macro defined above.

Definition at line 93 of file hex2otf.h.

## 5.3 src/unibdf2hex.c File Reference

unibdf2hex - Convert a BDF file into a unifont.hex file

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
Include dependency graph for unibdf2hex.c:




### Macros

- #define UNISTART 0x3400
  First Unicode code point to examine.
- #define UNISTOP 0x4DBF
  Last Unicode code point to examine.
- #define MAXBUF 256
  Maximum allowable input file line length - 1.

### Functions

- int main ()
  The main function.

### 5.3.1 Detailed Description

unibdf2hex - Convert a BDF file into a unifont.hex file

Author

Paul Hardy, January 2008

Copyright

Copyright (C) 2008, 2013 Paul Hardy

Note: currently this has hard-coded code points for glyphs extracted from Wen Quan Yi to create the Unifont source file "wqy.hex".

## 5.3.2 Function Documentation

### 5.3.2.1 main()

int main ( )

The main function.

Returns

Exit status is always 0 (successful termination).

Definition at line 46 of file unibdf2hex.c.

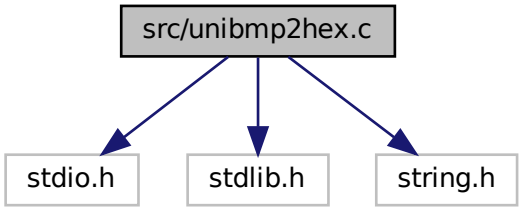
```
47 {
48    int i;
49    int digitsout;  /* how many hex digits we output in a bitmap */
50    int thispoint;
51    char inbuf[MAXBUF];
52    int bbxx, bbxy, bbxxoff, bbxyoff;
53
54    int descent=4; /* font descent wrt baseline */
55    int startrow;  /* row to start glyph        */
56    unsigned rowout;
57
58    while (fgets (inbuf, MAXBUF - 1, stdin) != NULL) {
59       if (strncmp (inbuf, "ENCODING ", 9) == 0) {
60          sscanf (&inbuf[9], "%d", &thispoint); /* get code point */
61          /*
62 If we want this code point, get the BBX (bounding box) and
63 BITMAP information.
64 */
65          if ((thispoint >= 0x2E80 && thispoint <= 0x2EFF) || // CJK Radicals Supplement
66              (thispoint >= 0x2F00 && thispoint <= 0x2FDF) || // Kangxi Radicals
67              (thispoint >= 0x2FF0 && thispoint <= 0x2FFF) || // Ideographic Description Characters
68              (thispoint >= 0x3001 && thispoint <= 0x303F) || // CJK Symbols and Punctuation (U+3000 is a space)
69              (thispoint >= 0x3100 && thispoint <= 0x312F) || // Bopomofo
70              (thispoint >= 0x31A0 && thispoint <= 0x31BF) || // Bopomofo extend
71              (thispoint >= 0x31C0 && thispoint <= 0x31EF) || // CJK Strokes
72              (thispoint >= 0x3400 && thispoint <= 0x4DBF) || // CJK Unified Ideographs Extension A
73              (thispoint >= 0x4E00 && thispoint <= 0x9FCF) || // CJK Unified Ideographs
74              (thispoint >= 0xF900 && thispoint <= 0xFAFF))   // CJK Compatibility Ideographs
75             {
76             while (fgets (inbuf, MAXBUF - 1, stdin) != NULL &&
77                    strncmp (inbuf, "BBX ", 4) != 0); /* find bounding box */
78
79             sscanf (&inbuf[4], "%d %d %d %d", &bbxx, &bbxy, &bbxxoff, &bbxyoff);
80             while (fgets (inbuf, MAXBUF - 1, stdin) != NULL &&
81                    strncmp (inbuf, "BITMAP", 6) != 0); /* find bitmap start */
82             fprintf (stdout, "%04X:", thispoint);
83             digitsout = 0;
84             /* Print initial blank rows */
85             startrow = descent + bbxyoff + bbxy;
86
87             /* Force everything to 16 pixels wide */
88             for (i = 16; i > startrow; i--) {
89                fprintf (stdout,"0000");
90                digitsout += 4;
91             }
92             while (fgets (inbuf, MAXBUF - 1, stdin) != NULL &&
93                    strncmp (inbuf, "END", 3) != 0) { /* copy bitmap until END */
94                sscanf (inbuf, "%X", &rowout);
95                /* Now force glyph to a 16x16 grid even if they'd fit in 8x16 */
96                if (bbxx <= 8) rowout «= 8;  /* shift left for 16x16 glyph */
97                rowout »= bbxxoff;
98                fprintf (stdout, "%04X", rowout);
99                digitsout += 4;
100            }
101
```

```
102             /* Pad for 16x16 glyph */
103             while (digitsout < 64) {
104               fprintf (stdout,"0000");
105               digitsout += 4;
106             }
107             fprintf (stdout,"\n");
108           }
109       }
110     }
111     exit (0);
112 }
```

## 5.4 src/unibmp2hex.c File Reference

unibmp2hex - Turn a .bmp or .wbmp glyph matrix into a GNU Unifont hex glyph set of 256 characters

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
Include dependency graph for unibmp2hex.c:




### Macros

- #define MAXBUF 256

  Maximum input file line length - 1.

### Functions

- int main (int argc, char *argv[])

  The main function.

## Variables

- unsigned hexdigit [16][4]

  32 bit representation of 16x8 0..F bitmap
- unsigned uniplane =0

  Unicode plane number, 0..0xff ff ff.
- unsigned planeset =0

  =1: use plane specified with -p parameter
- unsigned flip =0

  =1 if we're transposing glyph matrix
- unsigned forcewide =0

  =1 to set each glyph to 16 pixels wide
- unsigned unidigit [6][4]
- struct {
  char filetype [2]
  int file_size
  int image_offset
  int info_size
  int width
  int height
  int nplanes
  int bits_per_pixel
  int compression
  int image_size
  int x_ppm
  int y_ppm
  int ncolors
  int important_colors
  } bmp_header

- unsigned char color_table [256][4]

### 5.4.1 Detailed Description

unibmp2hex - Turn a .bmp or .wbmp glyph matrix into a GNU Unifont hex glyph set of 256 characters

Author

Paul Hardy, unifoundry <at> unifoundry.com, December 2007

Copyright



Synopsis: unibmp2hex [-iin_file.bmp] [-oout_file.hex] [-phex_page_num] [-w]

### 5.4.2 Function Documentation

#### 5.4.2.1 main()

```
int main (
                int argc,
                char * argv[] )
```

The main function.

Parameters

| | | |
|---|---|---|
| in | argc | The count of command line arguments. |
| in | argv | Pointer to array of command line arguments. |

Returns

This program exits with status 0.

Definition at line 149 of file unibmp2hex.c.

```
150 {
151
152   int i, j, k;              /* loop variables                        */
153   unsigned char inchar;       /* temporary input character */
154   char header[MAXBUF];        /* input buffer for bitmap file header */
155   int wbmp=0; /* =0 for Windows Bitmap (.bmp); 1 for Wireless Bitmap (.wbmp) */
156   int fatal; /* =1 if a fatal error occurred */
157   int match; /* =1 if we're still matching a pattern, 0 if no match */
158   int empty1, empty2; /* =1 if bytes tested are all zeroes */
159   unsigned char thischar1[16], thischar2[16]; /* bytes of hex char */
160   unsigned char thischar0[16], thischar3[16]; /* bytes for quadruple-width */
161   int thisrow; /* index to point into thischar1[] and thischar2[] */
162   int tmpsum;  /* temporary sum to see if a character is blank */
163   unsigned this_pixel;  /* color of one pixel, if > 1 bit per pixel */
164   unsigned next_pixels; /* pending group of 8 pixels being read */
165   unsigned color_mask = 0x00;  /* to invert monochrome bitmap, set to 0xFF */
166
167   unsigned char bitmap[17*32][18*32/8]; /* final bitmap */
168   /* For wide array:
169 0 = don't force glyph to double-width;
170 1 = force glyph to double-width;
171 4 = force glyph to quadruple-width.
172 */
173   char wide[0x200000]={0x200000 * 0};
174
175   char *infile="", *outfile="";  /* names of input and output files */
176   FILE *infp, *outfp;       /* file pointers of input and output files */
177
178   if (argc > 1) {
179     for (i = 1; i < argc; i++) {
180       if (argv[i][0] == '-') {  /* this is an option argument */
181         switch (argv[i][1]) {
182           case 'i':  /* name of input file */
183             infile = &argv[i][2];
184             break;
185           case 'o':  /* name of output file */
186             outfile = &argv[i][2];
187             break;
188           case 'p':  /* specify a Unicode plane */
189             sscanf (&argv[i][2], "%x", &uniplane); /* Get Unicode plane */
190             planeset = 1; /* Use specified range, not what's in bitmap */
191             break;
192           case 'w': /* force wide (16 pixels) for each glyph */
193             forcewide = 1;
194             break;
195           default:    /* if unrecognized option, print list and exit */
196             fprintf (stderr, "\nSyntax:\n\n");
197             fprintf (stderr, "    %s -p<Unicode_Page> ", argv[0]);
198             fprintf (stderr, "-i<Input_File> -o<Output_File> -w\n\n");
199             fprintf (stderr, "    -w specifies .wbmp output instead of ");
```

```
200                 fprintf (stderr, "default Windows .bmp output.\n\n");
201                 fprintf (stderr, "    -p is followed by 1 to 6 ");
202                 fprintf (stderr, "Unicode plane hex digits ");
203                 fprintf (stderr, "(default is Page 0).\n\n");
204                 fprintf (stderr, "\nExample:\n\n");
205                 fprintf (stderr, "    %s -p83 -iunifont.hex -ou83.bmp\n\n\n",
206                       argv[0]);
207                 exit (1);
208               }
209           }
210       }
211     }
212     /*
213  Make sure we can open any I/O files that were specified before
214  doing anything else.
215  */
216     if (strlen (infile) > 0) {
217       if ((infp = fopen (infile, "r")) == NULL) {
218         fprintf (stderr, "Error: can't open %s for input.\n", infile);
219         exit (1);
220       }
221     }
222     else {
223       infp = stdin;
224     }
225     if (strlen (outfile) > 0) {
226       if ((outfp = fopen (outfile, "w")) == NULL) {
227         fprintf (stderr, "Error: can't open %s for output.\n", outfile);
228         exit (1);
229       }
230     }
231     else {
232       outfp = stdout;
233     }
234     /*
235  Initialize selected code points for double width (16x16).
236  Double-width is forced in cases where a glyph (usually a combining
237  glyph) only occupies the left-hand side of a 16x16 grid, but must
238  be rendered as double-width to appear properly with other glyphs
239  in a given script.  If additions were made to a script after
240  Unicode 5.0, the Unicode version is given in parentheses after
241  the script name.
242  */
243     for (i = 0x0700; i <= 0x074F; i++) wide[i] = 1; /* Syriac                */
244     for (i = 0x0800; i <= 0x083F; i++) wide[i] = 1; /* Samaritan (5.2)       */
245     for (i = 0x0900; i <= 0x0DFF; i++) wide[i] = 1; /* Indic                 */
246     for (i = 0x1000; i <= 0x109F; i++) wide[i] = 1; /* Myanmar               */
247     for (i = 0x1100; i <= 0x11FF; i++) wide[i] = 1; /* Hangul Jamo           */
248     for (i = 0x1400; i <= 0x167F; i++) wide[i] = 1; /* Canadian Aboriginal   */
249     for (i = 0x1700; i <= 0x171F; i++) wide[i] = 1; /* Tagalog               */
250     for (i = 0x1720; i <= 0x173F; i++) wide[i] = 1; /* Hanunoo               */
251     for (i = 0x1740; i <= 0x175F; i++) wide[i] = 1; /* Buhid                 */
252     for (i = 0x1760; i <= 0x177F; i++) wide[i] = 1; /* Tagbanwa              */
253     for (i = 0x1780; i <= 0x17FF; i++) wide[i] = 1; /* Khmer                 */
254     for (i = 0x18B0; i <= 0x18FF; i++) wide[i] = 1; /* Ext. Can. Aboriginal  */
255     for (i = 0x1800; i <= 0x18AF; i++) wide[i] = 1; /* Mongolian             */
256     for (i = 0x1900; i <= 0x194F; i++) wide[i] = 1; /* Limbu                 */
257  // for (i = 0x1980; i <= 0x19DF; i++) wide[i] = 1; /* New Tai Lue           */
258     for (i = 0x1A00; i <= 0x1A1F; i++) wide[i] = 1; /* Buginese              */
259     for (i = 0x1A20; i <= 0x1AAF; i++) wide[i] = 1; /* Tai Tham (5.2)        */
260     for (i = 0x1B00; i <= 0x1B7F; i++) wide[i] = 1; /* Balinese              */
261     for (i = 0x1B80; i <= 0x1BBF; i++) wide[i] = 1; /* Sundanese (5.1)       */
262     for (i = 0x1BC0; i <= 0x1BFF; i++) wide[i] = 1; /* Batak (6.0)           */
263     for (i = 0x1C00; i <= 0x1C4F; i++) wide[i] = 1; /* Lepcha (5.1)          */
264     for (i = 0x1CC0; i <= 0x1CCF; i++) wide[i] = 1; /* Sundanese Supplement  */
265     for (i = 0x1CD0; i <= 0x1CFF; i++) wide[i] = 1; /* Vedic Extensions (5.2) */
266     wide[0x2329] = wide[0x232A] = 1; /* Left- & Right-pointing Angle Brackets */
267     for (i = 0x2E80; i <= 0xA4CF; i++) wide[i] = 1; /* CJK                   */
268  // for (i = 0x9FD8; i <= 0x9FE9; i++) wide[i] = 4; /* CJK quadruple-width   */
269     for (i = 0xA900; i <= 0xA92F; i++) wide[i] = 1; /* Kayah Li (5.1)        */
270     for (i = 0xA930; i <= 0xA95F; i++) wide[i] = 1; /* Rejang (5.1)          */
271     for (i = 0xA960; i <= 0xA97F; i++) wide[i] = 1; /* Hangul Jamo Extended-A */
272     for (i = 0xA980; i <= 0xA9DF; i++) wide[i] = 1; /* Javanese (5.2)        */
273     for (i = 0xAA00; i <= 0xAA5F; i++) wide[i] = 1; /* Cham (5.1)            */
274     for (i = 0xA9E0; i <= 0xA9FF; i++) wide[i] = 1; /* Myanmar Extended-B    */
275     for (i = 0xAA00; i <= 0xAA5F; i++) wide[i] = 1; /* Cham                  */
276     for (i = 0xAA60; i <= 0xAA7F; i++) wide[i] = 1; /* Myanmar Extended-A    */
277     for (i = 0xAAE0; i <= 0xAAFF; i++) wide[i] = 1; /* Meetei Mayek Ext (6.0) */
278     for (i = 0xABC0; i <= 0xABFF; i++) wide[i] = 1; /* Meetei Mayek (5.2)    */
279     for (i = 0xAC00; i <= 0xD7AF; i++) wide[i] = 1; /* Hangul Syllables      */
280     for (i = 0xD7B0; i <= 0xD7FF; i++) wide[i] = 1; /* Hangul Jamo Extended-B */
```

```
281    for (i = 0xF900; i <= 0xFAFF; i++) wide[i] = 1; /* CJK Compatibility      */
282    for (i = 0xFE10; i <= 0xFE1F; i++) wide[i] = 1; /* Vertical Forms        */
283    for (i = 0xFE30; i <= 0xFE60; i++) wide[i] = 1; /* CJK Compatibility Forms*/
284    for (i = 0xFFE0; i <= 0xFFE6; i++) wide[i] = 1; /* CJK Compatibility Forms*/
285
286    wide[0x303F] = 0; /* CJK half-space fill */
287
288    /* Supplemental Multilingual Plane (Plane 01) */
289    for (i = 0x010A00; i <= 0x010A5F; i++) wide[i] = 1; /* Kharoshthi         */
290    for (i = 0x011000; i <= 0x01107F; i++) wide[i] = 1; /* Brahmi             */
291    for (i = 0x011080; i <= 0x0110CF; i++) wide[i] = 1; /* Kaithi             */
292    for (i = 0x011100; i <= 0x01114F; i++) wide[i] = 1; /* Chakma             */
293    for (i = 0x011180; i <= 0x0111DF; i++) wide[i] = 1; /* Sharada            */
294    for (i = 0x011200; i <= 0x01124F; i++) wide[i] = 1; /* Khojki             */
295    for (i = 0x0112B0; i <= 0x0112FF; i++) wide[i] = 1; /* Khudawadi          */
296    for (i = 0x011300; i <= 0x01137F; i++) wide[i] = 1; /* Grantha            */
297    for (i = 0x011400; i <= 0x01147F; i++) wide[i] = 1; /* Newa               */
298    for (i = 0x011480; i <= 0x0114DF; i++) wide[i] = 1; /* Tirhuta            */
299    for (i = 0x011580; i <= 0x0115FF; i++) wide[i] = 1; /* Siddham            */
300    for (i = 0x011600; i <= 0x01165F; i++) wide[i] = 1; /* Modi               */
301    for (i = 0x011660; i <= 0x01167F; i++) wide[i] = 1; /* Mongolian Suppl.   */
302    for (i = 0x011680; i <= 0x0116CF; i++) wide[i] = 1; /* Takri              */
303    for (i = 0x011700; i <= 0x01173F; i++) wide[i] = 1; /* Ahom               */
304    for (i = 0x011800; i <= 0x01184F; i++) wide[i] = 1; /* Dogra              */
305    for (i = 0x011900; i <= 0x01195F; i++) wide[i] = 1; /* Dives Akuru        */
306    for (i = 0x0119A0; i <= 0x0119FF; i++) wide[i] = 1; /* Nandinagari        */
307    for (i = 0x011A00; i <= 0x011A4F; i++) wide[i] = 1; /* Zanabazar Square   */
308    for (i = 0x011A50; i <= 0x011AAF; i++) wide[i] = 1; /* Soyombo            */
309    for (i = 0x011B00; i <= 0x011B5F; i++) wide[i] = 1; /* Devanagari Extended-A*/
310    for (i = 0x011F00; i <= 0x011F5F; i++) wide[i] = 1; /* Kawi               */
311    for (i = 0x011C00; i <= 0x011C6F; i++) wide[i] = 1; /* Bhaiksuki          */
312    for (i = 0x011C70; i <= 0x011CBF; i++) wide[i] = 1; /* Marchen            */
313    for (i = 0x011D00; i <= 0x011D5F; i++) wide[i] = 1; /* Masaram Gondi      */
314    for (i = 0x011EE0; i <= 0x011EFF; i++) wide[i] = 1; /* Makasar            */
315    for (i = 0x012F90; i <= 0x012FFF; i++) wide[i] = 1; /* Cypro-Minoan       */
316    /* Make Bassa Vah all single width or all double width */
317    for (i = 0x016AD0; i <= 0x016AFF; i++) wide[i] = 1; /* Bassa Vah          */
318    for (i = 0x016B00; i <= 0x016B8F; i++) wide[i] = 1; /* Pahawh Hmong       */
319    for (i = 0x016F00; i <= 0x016F9F; i++) wide[i] = 1; /* Miao               */
320    for (i = 0x016FE0; i <= 0x016FFF; i++) wide[i] = 1; /* Ideograph Sym/Punct*/
321    for (i = 0x017000; i <= 0x0187FF; i++) wide[i] = 1; /* Tangut             */
322    for (i = 0x018800; i <= 0x018AFF; i++) wide[i] = 1; /* Tangut Components  */
323    for (i = 0x01AFF0; i <= 0x01AFFF; i++) wide[i] = 1; /* Kana Extended-B    */
324    for (i = 0x01B000; i <= 0x01B0FF; i++) wide[i] = 1; /* Kana Supplement    */
325    for (i = 0x01B100; i <= 0x01B12F; i++) wide[i] = 1; /* Kana Extended-A    */
326    for (i = 0x01B170; i <= 0x01B2FF; i++) wide[i] = 1; /* Nushu              */
327    for (i = 0x01CF00; i <= 0x01CFCF; i++) wide[i] = 1; /* Znamenny Musical   */
328    for (i = 0x01D100; i <= 0x01D1FF; i++) wide[i] = 1; /* Musical Symbols    */
329    for (i = 0x01D800; i <= 0x01DAAF; i++) wide[i] = 1; /* Sutton SignWriting */
330    for (i = 0x01E2C0; i <= 0x01E2FF; i++) wide[i] = 1; /* Wancho             */
331    for (i = 0x01E800; i <= 0x01E8DF; i++) wide[i] = 1; /* Mende Kikakui      */
332    for (i = 0x01F200; i <= 0x01F2FF; i++) wide[i] = 1; /* Encl Ideograp Suppl*/
333    wide[0x01F5E7] = 1;                                 /* Three Rays Right   */
334
335    /*
336 Determine whether or not the file is a Microsoft Windows Bitmap file.
337 If it starts with 'B', 'M', assume it's a Windows Bitmap file.
338 Otherwise, assume it's a Wireless Bitmap file.
339
340 WARNING: There isn't much in the way of error checking here --
341 if you give it a file that wasn't first created by hex2bmp.c,
342 all bets are off.
343 */
344    fatal = 0;  /* assume everything is okay with reading input file */
345    if ((header[0] = fgetc (infp)) != EOF) {
346       if ((header[1] = fgetc (infp)) != EOF) {
347          if (header[0] == 'B' && header[1] == 'M') {
348             wbmp = 0; /* Not a Wireless Bitmap -- it's a Windows Bitmap */
349          }
350          else {
351             wbmp = 1; /* Assume it's a Wireless Bitmap */
352          }
353       }
354       else
355          fatal = 1;
356    }
357    else
358       fatal = 1;
359
360    if (fatal) {
361       fprintf (stderr, "Fatal error; end of input file.\n\n");
```

```
362       exit (1);
363     }
364     /*
365  If this is a Wireless Bitmap (.wbmp) format file,
366  skip the header and point to the start of the bitmap itself.
367  */
368     if (wbmp) {
369       for (i=2; i<6; i++)
370         header[i] = fgetc (infp);
371       /*
372  Now read the bitmap.
373  */
374       for (i=0; i < 32*17; i++) {
375         for (j=0; j < 32*18/8; j++) {
376           inchar = fgetc (infp);
377           bitmap[i][j] = ~inchar;  /* invert bits for proper color */
378         }
379       }
380     }
381     /*
382  Otherwise, treat this as a Windows Bitmap file, because we checked
383  that it began with "BM".  Save the header contents for future use.
384  Expect a 14 byte standard BITMAPFILEHEADER format header followed
385  by a 40 byte standard BITMAPINFOHEADER Device Independent Bitmap
386  header, with data stored in little-endian format.
387  */
388     else {
389       for (i = 2; i < 54; i++)
390         header[i] = fgetc (infp);
391
392       bmp_header.filetype[0] = 'B';
393       bmp_header.filetype[1] = 'M';
394
395       bmp_header.file_size =
396           (header[2] & 0xFF)       | ((header[3] & 0xFF) «  8) |
397         ((header[4] & 0xFF) « 16) | ((header[5] & 0xFF) « 24);
398
399       /* header bytes 6..9 are reserved */
400
401       bmp_header.image_offset =
402           (header[10] & 0xFF)       | ((header[11] & 0xFF) «  8) |
403         ((header[12] & 0xFF) « 16) | ((header[13] & 0xFF) « 24);
404
405       bmp_header.info_size =
406           (header[14] & 0xFF)       | ((header[15] & 0xFF) «  8) |
407         ((header[16] & 0xFF) « 16) | ((header[17] & 0xFF) « 24);
408
409       bmp_header.width =
410           (header[18] & 0xFF)       | ((header[19] & 0xFF) «  8) |
411         ((header[20] & 0xFF) « 16) | ((header[21] & 0xFF) « 24);
412
413       bmp_header.height =
414           (header[22] & 0xFF)       | ((header[23] & 0xFF) «  8) |
415         ((header[24] & 0xFF) « 16) | ((header[25] & 0xFF) « 24);
416
417       bmp_header.nplanes =
418           (header[26] & 0xFF)       | ((header[27] & 0xFF) «  8);
419
420       bmp_header.bits_per_pixel =
421           (header[28] & 0xFF)       | ((header[29] & 0xFF) «  8);
422
423       bmp_header.compression =
424           (header[30] & 0xFF)       | ((header[31] & 0xFF) «  8) |
425         ((header[32] & 0xFF) « 16) | ((header[33] & 0xFF) « 24);
426
427       bmp_header.image_size =
428           (header[34] & 0xFF)       | ((header[35] & 0xFF) «  8) |
429         ((header[36] & 0xFF) « 16) | ((header[37] & 0xFF) « 24);
430
431       bmp_header.x_ppm =
432           (header[38] & 0xFF)       | ((header[39] & 0xFF) «  8) |
433         ((header[40] & 0xFF) « 16) | ((header[41] & 0xFF) « 24);
434
435       bmp_header.y_ppm =
436           (header[42] & 0xFF)       | ((header[43] & 0xFF) «  8) |
437         ((header[44] & 0xFF) « 16) | ((header[45] & 0xFF) « 24);
438
439       bmp_header.ncolors =
440           (header[46] & 0xFF)       | ((header[47] & 0xFF) «  8) |
441         ((header[48] & 0xFF) « 16) | ((header[49] & 0xFF) « 24);
442
```

```
443     bmp_header.important_colors =
444        (header[50] & 0xFF)       | ((header[51] & 0xFF) «  8) |
445       ((header[52] & 0xFF) « 16) | ((header[53] & 0xFF) « 24);
446
447     if (bmp_header.ncolors == 0)
448       bmp_header.ncolors = 1 « bmp_header.bits_per_pixel;
449
450     /* If a Color Table exists, read it */
451     if (bmp_header.ncolors > 0 && bmp_header.bits_per_pixel <= 8) {
452       for (i = 0; i < bmp_header.ncolors; i++) {
453         color_table[i][0] = fgetc (infp);  /* Red   */
454         color_table[i][1] = fgetc (infp);  /* Green */
455         color_table[i][2] = fgetc (infp);  /* Blue  */
456         color_table[i][3] = fgetc (infp);  /* Alpha */
457       }
458       /*
459 Determine from the first color table entry whether we
460 are inverting the resulting bitmap image.
461 */
462       if ( (color_table[0][0] + color_table[0][1] + color_table[0][2])
463           < (3 * 128) ) {
464         color_mask = 0xFF;
465       }
466     }
467
468 #ifdef DEBUG
469
470     /*
471 Print header info for possibly adding support for
472 additional file formats in the future, to determine
473 how the bitmap is encoded.
474 */
475     fprintf (stderr, "Filetype: '%c%c'\n",
476                bmp_header.filetype[0], bmp_header.filetype[1]);
477     fprintf (stderr, "File Size: %d\n", bmp_header.file_size);
478     fprintf (stderr, "Image Offset: %d\n", bmp_header.image_offset);
479     fprintf (stderr, "Info Header Size: %d\n", bmp_header.info_size);
480     fprintf (stderr, "Image Width: %d\n", bmp_header.width);
481     fprintf (stderr, "Image Height: %d\n", bmp_header.height);
482     fprintf (stderr, "Number of Planes: %d\n", bmp_header.nplanes);
483     fprintf (stderr, "Bits per Pixel: %d\n", bmp_header.bits_per_pixel);
484     fprintf (stderr, "Compression Method: %d\n", bmp_header.compression);
485     fprintf (stderr, "Image Size: %d\n", bmp_header.image_size);
486     fprintf (stderr, "X Pixels per Meter: %d\n", bmp_header.x_ppm);
487     fprintf (stderr, "Y Pixels per Meter: %d\n", bmp_header.y_ppm);
488     fprintf (stderr, "Number of Colors: %d\n", bmp_header.ncolors);
489     fprintf (stderr, "Important Colors: %d\n", bmp_header.important_colors);
490
491 #endif
492
493     /*
494 Now read the bitmap.
495 */
496     for (i = 32*17-1; i >= 0; i--) {
497       for (j=0; j < 32*18/8; j++) {
498         next_pixels = 0x00;  /* initialize next group of 8 pixels */
499         /* Read a monochrome image -- the original case */
500         if (bmp_header.bits_per_pixel == 1) {
501           next_pixels = fgetc (infp);
502         }
503         /* Read a 32 bit per pixel RGB image; convert to monochrome */
504         else if ( bmp_header.bits_per_pixel == 24 ||
505                   bmp_header.bits_per_pixel == 32) {
506           next_pixels = 0;
507           for (k = 0; k < 8; k++) {  /* get next 8 pixels */
508             this_pixel = (fgetc (infp) & 0xFF) +
509                          (fgetc (infp) & 0xFF) +
510                          (fgetc (infp) & 0xFF);
511
512             if (bmp_header.bits_per_pixel == 32) {
513               (void) fgetc (infp);  /* ignore alpha value */
514             }
515
516             /* convert RGB color space to monochrome */
517             if (this_pixel >= (128 * 3))
518               this_pixel = 0;
519             else
520               this_pixel = 1;
521
522             /* shift next pixel color into place for 8 pixels total */
523             next_pixels = (next_pixels « 1) | this_pixel;
```

```
524             }
525           }
526           if (bmp_header.height < 0) {  /* Bitmap drawn top to bottom */
527             bitmap [(32*17-1) - i] [j] = next_pixels;
528           }
529           else {  /* Bitmap drawn bottom to top */
530             bitmap [i][j] = next_pixels;
531           }
532         }
533       }
534
535       /*
536 If any bits are set in color_mask, apply it to
537 entire bitmap to invert black <--> white.
538 */
539       if (color_mask != 0x00) {
540         for (i = 32*17-1; i >= 0; i--) {
541           for (j=0; j < 32*18/8; j++) {
542             bitmap [i][j] ^= color_mask;
543           }
544         }
545       }
546
547     }
548
549     /*
550 We've read the entire file.  Now close the input file pointer.
551 */
552     fclose (infp);
553     /*
554 We now have the header portion in the header[] array,
555 and have the bitmap portion from top-to-bottom in the bitmap[] array.
556 */
557     /*
558 If no Unicode range (U+nnnnnn00 through U+nnnnnnFF) was specified
559 with a -p parameter, determine the range from the digits in the
560 bitmap itself.
561
562 Store bitmaps for the hex digit patterns that this file uses.
563 */
564     if (!planeset) {  /* If Unicode range not specified with -p parameter */
565       for (i = 0x0; i <= 0xF; i++) {  /* hex digit pattern we're storing */
566         for (j = 0; j < 4; j++) {
567           hexdigit[i][j]   =
568             ((unsigned)bitmap[32 * (i+1) + 4 * j + 8    ][6] « 24 ) |
569             ((unsigned)bitmap[32 * (i+1) + 4 * j + 8 + 1][6] « 16 ) |
570             ((unsigned)bitmap[32 * (i+1) + 4 * j + 8 + 2][6] «  8 ) |
571             ((unsigned)bitmap[32 * (i+1) + 4 * j + 8 + 3][6]       );
572         }
573       }
574       /*
575 Read the Unicode plane digits into arrays for comparison, to
576 determine the upper four hex digits of the glyph addresses.
577 */
578       for (i = 0; i < 4; i++) {
579         for (j = 0; j < 4; j++) {
580           unidigit[i][j] =
581             ((unsigned)bitmap[32 * 0 + 4 * j + 8 + 1][i + 3] « 24 ) |
582             ((unsigned)bitmap[32 * 0 + 4 * j + 8 + 2][i + 3] « 16 ) |
583             ((unsigned)bitmap[32 * 0 + 4 * j + 8 + 3][i + 3] «  8 ) |
584             ((unsigned)bitmap[32 * 0 + 4 * j + 8 + 4][i + 3]       );
585         }
586       }
587
588       tmpsum = 0;
589       for (i = 4; i < 6; i++) {
590         for (j = 0; j < 4; j++) {
591           unidigit[i][j] =
592             ((unsigned)bitmap[32 * 1 + 4 * j + 8    ][i] « 24 ) |
593             ((unsigned)bitmap[32 * 1 + 4 * j + 8 + 1][i] « 16 ) |
594             ((unsigned)bitmap[32 * 1 + 4 * j + 8 + 2][i] «  8 ) |
595             ((unsigned)bitmap[32 * 1 + 4 * j + 8 + 3][i]       );
596           tmpsum |= unidigit[i][j];
597         }
598       }
599       if (tmpsum == 0) {  /* the glyph matrix is transposed */
600         flip = 1;  /* note transposed order for processing glyphs in matrix */
601         /*
602 Get 5th and 6th hex digits by shifting first column header left by
603 1.5 columns, thereby shifting the hex digit right after the leading
604 "U+nnnn" page number.
```

```
605 */
606         for (i = 0x08; i < 0x18; i++) {
607           bitmap[i][7] = (bitmap[i][8] « 4) | ((bitmap[i][ 9] » 4) & 0xf);
608           bitmap[i][8] = (bitmap[i][9] « 4) | ((bitmap[i][10] » 4) & 0xf);
609         }
610         for (i = 4; i < 6; i++) {
611           for (j = 0; j < 4; j++) {
612             unidigit[i][j] =
613               ((unsigned)bitmap[4 * j + 8 + 1][i + 3] « 24 ) |
614               ((unsigned)bitmap[4 * j + 8 + 2][i + 3] « 16 ) |
615               ((unsigned)bitmap[4 * j + 8 + 3][i + 3] «  8 ) |
616               ((unsigned)bitmap[4 * j + 8 + 4][i + 3]       );
617           }
618         }
619       }
620 
621       /*
622 Now determine the Unicode plane by comparing unidigit[0..5] to
623 the hexdigit[0x0..0xF] array.
624 */
625       uniplane = 0;
626       for (i=0; i<6; i++) { /* go through one bitmap digit at a time */
627         match = 0; /* haven't found pattern yet */
628         for (j = 0x0; !match && j <= 0xF; j++) {
629           if (unidigit[i][0] == hexdigit[j][0] &&
630               unidigit[i][1] == hexdigit[j][1] &&
631               unidigit[i][2] == hexdigit[j][2] &&
632               unidigit[i][3] == hexdigit[j][3]) { /* we found the digit */
633             uniplane |= j;
634             match = 1;
635           }
636         }
637         uniplane «= 4;
638       }
639       uniplane »= 4;
640     }
641     /*
642 Now read each glyph and print it as hex.
643 */
644     for (i = 0x0; i <= 0xf; i++) {
645       for (j = 0x0; j <= 0xf; j++) {
646         for (k = 0; k < 16; k++) {
647           if (flip) {  /* transpose glyph matrix */
648             thischar0[k] = bitmap[32*(j+1) + k + 7][4 * (i+2)    ];
649             thischar1[k] = bitmap[32*(j+1) + k + 7][4 * (i+2) + 1];
650             thischar2[k] = bitmap[32*(j+1) + k + 7][4 * (i+2) + 2];
651             thischar3[k] = bitmap[32*(j+1) + k + 7][4 * (i+2) + 3];
652           }
653           else {
654             thischar0[k] = bitmap[32*(i+1) + k + 7][4 * (j+2)    ];
655             thischar1[k] = bitmap[32*(i+1) + k + 7][4 * (j+2) + 1];
656             thischar2[k] = bitmap[32*(i+1) + k + 7][4 * (j+2) + 2];
657             thischar3[k] = bitmap[32*(i+1) + k + 7][4 * (j+2) + 3];
658           }
659         }
660         /*
661 If the second half of the 16*16 character is all zeroes, this
662 character is only 8 bits wide, so print a half-width character.
663 */
664         empty1 = empty2 = 1;
665         for (k=0; (empty1 || empty2) && k < 16; k++) {
666           if (thischar1[k] != 0) empty1 = 0;
667           if (thischar2[k] != 0) empty2 = 0;
668           }
669         /*
670 Only print this glyph if it isn't blank.
671 */
672         if (!empty1 || !empty2) {
673           /*
674 If the second half is empty, this is a half-width character.
675 Only print the first half.
676 */
677           /*
678 Original GNU Unifont format is four hexadecimal digit character
679 code followed by a colon followed by a hex string.  Add support
680 for codes beyond the Basic Multilingual Plane.
681 
682 Unicode ranges from U+0000 to U+10FFFF, so print either a
683 4-digit or a 6-digit code point.  Note that this software
684 should support up to an 8-digit code point, extending beyond
685 the normal Unicode range, but this has not been fully tested.
```

```
686 */
687             if (uniplane > 0xff)
688                fprintf (outfp, "%04X%X%X:", uniplane, i, j); // 6 digit code pt.
689             else
690                fprintf (outfp, "%02X%X%X:", uniplane, i, j); // 4 digit code pt.
691             for (thisrow=0; thisrow<16; thisrow++) {
692                /*
693 If second half is empty and we're not forcing this
694 code point to double width, print as single width.
695 */
696                if (!forcewide &&
697                    empty2 && !wide[(uniplane « 8) | (i « 4) | j]) {
698                   fprintf (outfp,
699                            "%02X",
700                            thischar1[thisrow]);
701                }
702                else if (wide[(uniplane « 8) | (i « 4) | j] == 4) {
703                   /* quadruple-width; force 32nd pixel to zero */
704                   fprintf (outfp,
705                            "%02X%02X%02X%02X",
706                            thischar0[thisrow], thischar1[thisrow],
707                            thischar2[thisrow], thischar3[thisrow] & 0xFE);
708                }
709                else { /* treat as double-width */
710                   fprintf (outfp,
711                            "%02X%02X",
712                            thischar1[thisrow], thischar2[thisrow]);
713                }
714             }
715             fprintf (outfp, "\n");
716          }
717       }
718    }
719    exit (0);
720 }
```

### 5.4.3 Variable Documentation

#### 5.4.3.1 bmp_header

struct { ... } bmp_header

Bitmap Header parameters

#### 5.4.3.2 color_table

unsigned char color_table[256][4]

Bitmap Color Table – maximum of 256 colors in a BMP file

Definition at line 137 of file unibmp2hex.c.

#### 5.4.3.3 unidigit

unsigned unidigit[6][4]

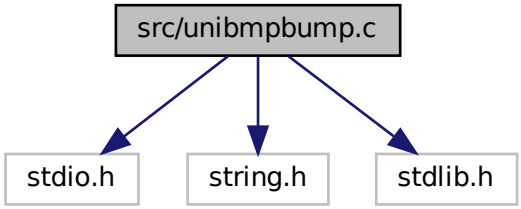
The six Unicode plane digits, from left-most (0) to right-most (5)

Definition at line 115 of file unibmp2hex.c.

## 5.5 src/unibmpbump.c File Reference

unibmpbump - Adjust a Microsoft bitmap (.bmp) file that was created by unihex2png but converted to .bmp

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
Include dependency graph for unibmpbump.c:




### Macros

- #define VERSION "1.0"
  Version of this program.
- #define MAX_COMPRESSION_METHOD 13
  Maximum supported compression method.

### Functions

- int main (int argc, char *argv[])
  The main function.
- unsigned get_bytes (FILE *infp, int nbytes)
  Get from 1 to 4 bytes, inclusive, from input file.
- void regrid (unsigned *image_bytes)
  After reading in the image, shift it.

### 5.5.1 Detailed Description

unibmpbump - Adjust a Microsoft bitmap (.bmp) file that was created by unihex2png but converted to .bmp

Author

Paul Hardy, unifoundry <at> unifoundry.com

Copyright



This program shifts the glyphs in a bitmap file to adjust an original PNG file that was saved in BMP format. This is so the result matches the format of a unihex2bmp image. This conversion then lets unibmp2hex decode the result.

Synopsis: unibmpbump [-iin_file.bmp] [-oout_file.bmp]

## 5.5.2 Function Documentation

### 5.5.2.1 get_bytes()

```
unsigned get_bytes (
                FILE * infp,
                int nbytes )
```

Get from 1 to 4 bytes, inclusive, from input file.

Parameters

| | | |
|---|---|---|
| in | infp | Pointer to input file. |
| in | nbytes | Number of bytes to read, from 1 to 4, inclusive. |

Returns

The unsigned 1 to 4 bytes in machine native endian format.

Definition at line 487 of file unibmpbump.c.

```
487                                     {
488     int i;
489     unsigned char inchar[4];
490     unsigned inword;
491
492     for (i = 0; i < nbytes; i++) {
493       if (fread (&inchar[i], 1, 1, infp) != 1) {
494         inchar[i] = 0;
495       }
496     }
497     for (i = nbytes; i < 4; i++) inchar[i] = 0;
498
499     inword = ((inchar[3] & 0xFF) << 24) | ((inchar[2] & 0xFF) << 16) |
500              ((inchar[1] & 0xFF) <<  8) |  (inchar[0] & 0xFF);
501
502     return inword;
503 }
```

#### 5.5.2.2 main()

```
int main (
                int argc,
                char * argv[] )
```

The main function.

Parameters

| | | |
|---|---|---|
| in | argc | The count of command line arguments. |
| in | argv | Pointer to array of command line arguments. |

Returns

This program exits with status EXIT_SUCCESS.

Definition at line 50 of file unibmpbump.c.

```
50                              {
51
52   /*
53 Values preserved from file header (first 14 bytes).
54 */
55   char file_format[3];       /* "BM" for original Windows format          */
56   unsigned filesize;        /* size of file in bytes                     */
57   unsigned char rsvd_hdr[4]; /* 4 reserved bytes                         */
58   unsigned image_start;      /* byte offset of image in file             */
59
60   /*
61 Values preserved from Device Independent Bitmap (DIB) Header.
62
63 The DIB fields below are in the standard 40-byte header.  Version
64 4 and version 5 headers have more information, mainly for color
65 information.  That is skipped over, because a valid glyph image
66 is just monochrome.
67 */
68   int dib_length;           /* in bytes, for parsing by header version   */
69   int image_width = 0;      /* Signed image width                        */
70   int image_height = 0;     /* Signed image height                       */
71   int num_planes = 0;       /* number of planes; must be 1               */
72   int bits_per_pixel = 0;   /* for palletized color maps (< 2^16 colors) */
73   /*
74 The following fields are not in the original spec, so initialize
75 them to 0 so we can correctly parse an original file format.
76 */
77   int compression_method=0;  /* 0 --> uncompressed RGB/monochrome        */
78   int image_size = 0;        /* 0 is a valid size if no compression      */
79   int hres = 0;             /* image horizontal resolution               */
80   int vres = 0;             /* image vertical resolution                 */
81   int num_colors = 0;       /* Number of colors for pallettized images   */
82   int important_colors = 0;  /* Number of significant colors (0 or 2)    */
83
84   int true_colors = 0;      /* interpret num_colors, which can equal 0   */
85
86   /*
87 Color map.  This should be a monochrome file, so only two
88 colors are stored.
89 */
90   unsigned char color_map[2][4]; /* two of R, G, B, and possibly alpha  */
91
92   /*
93 The monochrome image bitmap, stored as a vector 544 rows by
94 72*8 columns.
95 */
96   unsigned image_bytes[544*72];
97
98   /*
99 Flags for conversion & I/O.
```

```
100 */
101   int verbose      = 0;      /* Whether to print file info on stderr     */
102   unsigned image_xor = 0x00; /* Invert (= 0xFF) if color 0 is not black  */
103
104   /*
105 Temporary variables.
106 */
107   int i, j, k;           /* loop variables */
108
109   /* Compression type, for parsing file */
110   char *compression_type[MAX_COMPRESSION_METHOD + 1] = {
111     "BI_RGB",            /*  0 */
112     "BI_RLE8",           /*  1 */
113     "BI_RLE4",           /*  2 */
114     "BI_BITFIELDS",      /*  3 */
115     "BI_JPEG",           /*  4 */
116     "BI_PNG",            /*  5 */
117     "BI_ALPHABITFIELDS", /*  6 */
118     "", "", "", "",      /*  7 - 10 */
119     "BI_CMYK",           /* 11 */
120     "BI_CMYKRLE8",       /* 12 */
121     "BI_CMYKRLE4",       /* 13 */
122   };
123
124   /* Standard unihex2bmp.c header for BMP image */
125   unsigned standard_header [62] = {
126     /*  0 */ 0x42, 0x4d, 0x3e, 0x99, 0x00, 0x00, 0x00, 0x00,
127     /*  8 */ 0x00, 0x00, 0x3e, 0x00, 0x00, 0x00, 0x28, 0x00,
128     /* 16 */ 0x00, 0x00, 0x40, 0x02, 0x00, 0x00, 0x20, 0x02,
129     /* 24 */ 0x00, 0x00, 0x01, 0x00, 0x01, 0x00, 0x00, 0x00,
130     /* 32 */ 0x00, 0x00, 0x00, 0x99, 0x00, 0x00, 0xc4, 0x0e,
131     /* 40 */ 0x00, 0x00, 0xc4, 0x0e, 0x00, 0x00, 0x00, 0x00,
132     /* 48 */ 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
133     /* 56 */ 0x00, 0x00, 0xff, 0xff, 0xff, 0x00
134   };
135
136   unsigned get_bytes (FILE *, int);
137   void     regrid    (unsigned *);
138
139   char *infile="", *outfile="";  /* names of input and output files         */
140   FILE *infp, *outfp;            /* file pointers of input and output files */
141
142   /*
143 Process command line arguments.
144 */
145   if (argc > 1) {
146     for (i = 1; i < argc; i++) {
147       if (argv[i][0] == '-') {  /* this is an option argument */
148         switch (argv[i][1]) {
149           case 'i':  /* name of input file */
150             infile = &argv[i][2];
151             break;
152           case 'o':  /* name of output file */
153             outfile = &argv[i][2];
154             break;
155           case 'v':  /* verbose output */
156             verbose = 1;
157             break;
158           case 'V':  /* print version & quit */
159             fprintf (stderr, "unibmpbump version %s\n\n", VERSION);
160             exit (EXIT_SUCCESS);
161             break;
162           case '-':  /* see if "--verbose" */
163             if (strcmp (argv[i], "--verbose") == 0) {
164               verbose = 1;
165             }
166             else if (strcmp (argv[i], "--version") == 0) {
167               fprintf (stderr, "unibmpbump version %s\n\n", VERSION);
168               exit (EXIT_SUCCESS);
169             }
170             break;
171           default:   /* if unrecognized option, print list and exit */
172             fprintf (stderr, "\nSyntax:\n\n");
173             fprintf (stderr, "    unibmpbump ");
174             fprintf (stderr, "-i<Input_File> -o<Output_File>\n\n");
175             fprintf (stderr, "-v or --verbose gives verbose output");
176             fprintf (stderr, " on stderr\n\n");
177             fprintf (stderr, "-V or --version prints version");
178             fprintf (stderr, " on stderr and exits\n\n");
179             fprintf (stderr, "\nExample:\n\n");
180             fprintf (stderr, "    unibmpbump -iuni0101.bmp");
```

```
181                 fprintf (stderr, " -onew-uni0101.bmp\n\n");
182                 exit (EXIT_SUCCESS);
183               }
184             }
185           }
186         }
187
188         /*
189 Make sure we can open any I/O files that were specified before
190 doing anything else.
191 */
192         if (strlen (infile) > 0) {
193           if ((infp = fopen (infile, "r")) == NULL) {
194             fprintf (stderr, "Error: can't open %s for input.\n", infile);
195             exit (EXIT_FAILURE);
196           }
197         }
198         else {
199           infp = stdin;
200         }
201         if (strlen (outfile) > 0) {
202           if ((outfp = fopen (outfile, "w")) == NULL) {
203             fprintf (stderr, "Error: can't open %s for output.\n", outfile);
204             exit (EXIT_FAILURE);
205           }
206         }
207         else {
208           outfp = stdout;
209         }
210
211
212         /* Read bitmap file header */
213         file_format[0] = get_bytes (infp, 1);
214         file_format[1] = get_bytes (infp, 1);
215         file_format[2] = '\0';  /* Terminate string with null */
216
217         /* Read file size */
218         filesize = get_bytes (infp, 4);
219
220         /* Read Reserved bytes */
221         rsvd_hdr[0] = get_bytes (infp, 1);
222         rsvd_hdr[1] = get_bytes (infp, 1);
223         rsvd_hdr[2] = get_bytes (infp, 1);
224         rsvd_hdr[3] = get_bytes (infp, 1);
225
226         /* Read Image Offset Address within file */
227         image_start = get_bytes (infp, 4);
228
229         /*
230 See if this looks like a valid image file based on
231 the file header first two bytes.
232 */
233         if (strncmp (file_format, "BM", 2) != 0) {
234           fprintf (stderr, "\nInvalid file format: not file type \"BM\".\n\n");
235           exit (EXIT_FAILURE);
236         }
237
238         if (verbose) {
239           fprintf (stderr, "\nFile Header:\n");
240           fprintf (stderr, "    File Type:     \"%s\"\n", file_format);
241           fprintf (stderr, "    File Size:     %d bytes\n", filesize);
242           fprintf (stderr, "    Reserved:      ");
243           for (i = 0; i < 4; i++) fprintf (stderr, " 0x%02X", rsvd_hdr[i]);
244           fputc ('\n', stderr);
245           fprintf (stderr, "    Image Start: %d. = 0x%02X = 0%05o\n\n",
246                    image_start, image_start, image_start);
247         }  /* if (verbose) */
248
249         /*
250 Device Independent Bitmap (DIB) Header: bitmap information header
251 ("BM" format file DIB Header is 12 bytes long).
252 */
253         dib_length = get_bytes (infp, 4);
254
255         /*
256 Parse one of three versions of Device Independent Bitmap (DIB) format:
257
258 Length  Format
259 ------  ------
260 12    BITMAPCOREHEADER
261 40    BITMAPINFOHEADER
```

```
262 108   BITMAPV4HEADER
263 124   BITMAPV5HEADER
264 */
265   if (dib_length == 12) { /* BITMAPCOREHEADER format -- UNTESTED */
266     image_width    = get_bytes (infp, 2);
267     image_height   = get_bytes (infp, 2);
268     num_planes     = get_bytes (infp, 2);
269     bits_per_pixel = get_bytes (infp, 2);
270   }
271   else if (dib_length >= 40) { /* BITMAPINFOHEADER format or later */
272     image_width = get_bytes (infp, 4);
273     image_height       = get_bytes (infp, 4);
274     num_planes         = get_bytes (infp, 2);
275     bits_per_pixel     = get_bytes (infp, 2);
276     compression_method = get_bytes (infp, 4);  /* BI_BITFIELDS */
277     image_size         = get_bytes (infp, 4);
278     hres               = get_bytes (infp, 4);
279     vres               = get_bytes (infp, 4);
280     num_colors         = get_bytes (infp, 4);
281     important_colors   = get_bytes (infp, 4);
282
283     /* true_colors is true number of colors in image */
284     if (num_colors == 0)
285       true_colors = 1 « bits_per_pixel;
286     else
287       true_colors = num_colors;
288
289     /*
290 If dib_length > 40, the format is BITMAPV4HEADER or
291 BITMAPV5HEADER.  As this program is only designed
292 to handle a monochrome image, we can ignore the rest
293 of the header but must read past the remaining bytes.
294 */
295     for (i = 40; i < dib_length; i++) (void)get_bytes (infp, 1);
296   }
297
298   if (verbose) {
299     fprintf (stderr, "Device Independent Bitmap (DIB) Header:\n");
300     fprintf (stderr, "   DIB Length:  %9d bytes (version = ", dib_length);
301
302     if      (dib_length ==  12) fprintf (stderr, "\"BITMAPCOREHEADER\")\n");
303     else if (dib_length ==  40) fprintf (stderr, "\"BITMAPINFOHEADER\")\n");
304     else if (dib_length == 108) fprintf (stderr, "\"BITMAPV4HEADER\")\n");
305     else if (dib_length == 124) fprintf (stderr, "\"BITMAPV5HEADER\")\n");
306     else fprintf (stderr, "unknown)");
307     fprintf (stderr, "   Bitmap Width:   %6d pixels\n", image_width);
308     fprintf (stderr, "   Bitmap Height:  %6d pixels\n", image_height);
309     fprintf (stderr, "   Color Planes:   %6d\n",        num_planes);
310     fprintf (stderr, "   Bits per Pixel: %6d\n",        bits_per_pixel);
311     fprintf (stderr, "   Compression Method: %2d --> ", compression_method);
312     if (compression_method <= MAX_COMPRESSION_METHOD) {
313       fprintf (stderr, "%s", compression_type [compression_method]);
314     }
315     /*
316 Supported compression method values:
317 0 --> uncompressed RGB
318 11 --> uncompressed CMYK
319 */
320     if (compression_method == 0 || compression_method == 11) {
321       fprintf (stderr, " (no compression)");
322     }
323     else {
324       fprintf (stderr, "Image uses compression; this is unsupported.\n\n");
325       exit (EXIT_FAILURE);
326     }
327     fprintf (stderr, "\n");
328     fprintf (stderr, "   Image Size:               %5d bytes\n", image_size);
329     fprintf (stderr, "   Horizontal Resolution: %5d pixels/meter\n", hres);
330     fprintf (stderr, "   Vertical Resolution:   %5d pixels/meter\n", vres);
331     fprintf (stderr, "   Number of Colors:       %5d", num_colors);
332     if (num_colors != true_colors) {
333       fprintf (stderr, " --> %d", true_colors);
334     }
335     fputc ('\n', stderr);
336     fprintf (stderr, "   Important Colors:       %5d", important_colors);
337     if (important_colors == 0)
338       fprintf (stderr, " (all colors are important)");
339     fprintf (stderr, "\n\n");
340   }  /* if (verbose) */
341
342   /*
```

```
343 Print Color Table information for images with pallettized colors.
344 */
345   if (bits_per_pixel <= 8) {
346     for (i = 0; i < 2; i++) {
347       color_map [i][0] = get_bytes (infp, 1);
348       color_map [i][1] = get_bytes (infp, 1);
349       color_map [i][2] = get_bytes (infp, 1);
350       color_map [i][3] = get_bytes (infp, 1);
351     }
352     /* Skip remaining color table entries if more than 2 */
353     while (i < true_colors) {
354       (void) get_bytes (infp, 4);
355       i++;
356     }
357
358     if (color_map [0][0] >= 128) image_xor = 0xFF;  /* Invert colors */
359   }
360
361   if (verbose) {
362     fprintf (stderr, "Color Palette [R, G, B, %s] Values:\n",
363             (dib_length <= 40) ? "reserved" : "Alpha");
364     for (i = 0; i < 2; i++) {
365       fprintf (stderr, "%7d: [", i);
366       fprintf (stderr, "%3d,",   color_map [i][0] & 0xFF);
367       fprintf (stderr, "%3d,",   color_map [i][1] & 0xFF);
368       fprintf (stderr, "%3d,",   color_map [i][2] & 0xFF);
369       fprintf (stderr, "%3d]\n", color_map [i][3] & 0xFF);
370     }
371     if (image_xor == 0xFF) fprintf (stderr, "Will Invert Colors.\n");
372     fputc ('\n', stderr);
373
374   }  /* if (verbose) */
375
376
377   /*
378 Check format before writing output file.
379 */
380   if (image_width != 560 && image_width != 576) {
381     fprintf (stderr, "\nUnsupported image width: %d\n", image_width);
382     fprintf (stderr, "Width should be 560 or 576 pixels.\n\n");
383     exit (EXIT_FAILURE);
384   }
385
386   if (image_height != 544) {
387     fprintf (stderr, "\nUnsupported image height: %d\n", image_height);
388     fprintf (stderr, "Height should be 544 pixels.\n\n");
389     exit (EXIT_FAILURE);
390   }
391
392   if (num_planes != 1) {
393     fprintf (stderr, "\nUnsupported number of planes: %d\n", num_planes);
394     fprintf (stderr, "Number of planes should be 1.\n\n");
395     exit (EXIT_FAILURE);
396   }
397
398   if (bits_per_pixel != 1) {
399     fprintf (stderr, "\nUnsupported number of bits per pixel: %d\n",
400             bits_per_pixel);
401     fprintf (stderr, "Bits per pixel should be 1.\n\n");
402     exit (EXIT_FAILURE);
403   }
404
405   if (compression_method != 0 && compression_method != 11) {
406     fprintf (stderr, "\nUnsupported compression method: %d\n",
407             compression_method);
408     fprintf (stderr, "Compression method should be 1 or 11.\n\n");
409     exit (EXIT_FAILURE);
410   }
411
412   if (true_colors != 2) {
413     fprintf (stderr, "\nUnsupported number of colors: %d\n", true_colors);
414     fprintf (stderr, "Number of colors should be 2.\n\n");
415     exit (EXIT_FAILURE);
416   }
417
418
419   /*
420 If we made it this far, things look okay, so write out
421 the standard header for image conversion.
422 */
423   for (i = 0; i < 62; i++) fputc (standard_header[i], outfp);
```

```
424
425
426     /*
427 Image Data.  Each row must be a multiple of 4 bytes, with
428 padding at the end of each row if necessary.
429 */
430     k = 0;  /* byte number within the binary image */
431     for (i = 0; i < 544; i++) {
432       /*
433 If original image is 560 pixels wide (not 576), add
434 2 white bytes at beginning of row.
435 */
436       if (image_width == 560) {  /* Insert 2 white bytes */
437         image_bytes[k++] = 0xFF;
438         image_bytes[k++] = 0xFF;
439       }
440       for (j = 0; j < 70; j++) {  /* Copy next 70 bytes */
441         image_bytes[k++] = (get_bytes (infp, 1) & 0xFF) ^ image_xor;
442       }
443       /*
444 If original image is 560 pixels wide (not 576), skip
445 2 padding bytes at end of row in file because we inserted
446 2 white bytes at the beginning of the row.
447 */
448       if (image_width == 560) {
449         (void) get_bytes (infp, 2);
450       }
451       else {  /* otherwise, next 2 bytes are part of the image so copy them */
452         image_bytes[k++] = (get_bytes (infp, 1) & 0xFF) ^ image_xor;
453         image_bytes[k++] = (get_bytes (infp, 1) & 0xFF) ^ image_xor;
454       }
455     }
456
457
458     /*
459 Change the image to match the unihex2bmp.c format if original wasn't
460 */
461     if (image_width == 560) {
462       regrid (image_bytes);
463     }
464
465     for (i = 0; i < 544 * 576 / 8; i++) {
466       fputc (image_bytes[i], outfp);
467     }
468
469
470     /*
471 Wrap up.
472 */
473     fclose (infp);
474     fclose (outfp);
475
476     exit (EXIT_SUCCESS);
477 }
```

#### 5.5.2.3 regrid()

```
void regrid (
                unsigned * image_bytes )
```

After reading in the image, shift it.

This function adjusts the input image from an original PNG file to match unihex2bmp.c format.

Parameters

| | | |
|---|---|---|
| in,out | image_bytes | The pixels in an image. |

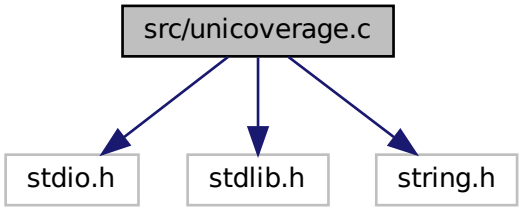
Definition at line 514 of file unibmpbump.c.

```
514                                 {
515   int i, j, k;  /* loop variables */
516   int offset;
517   unsigned glyph_row; /* one grid row of 32 pixels */
518   unsigned last_pixel; /* last pixel in a byte, to preserve */
519
520   /* To insert "00" after "U+" at top of image */
521   char zero_pattern[16] = {
522       0x00, 0x00, 0x00, 0x00, 0x18, 0x24, 0x42, 0x42,
523       0x42, 0x42, 0x42, 0x42, 0x24, 0x18, 0x00, 0x00
524   };
525
526   /* This is the horizontal grid pattern on glyph boundaries */
527   unsigned hgrid[72] = {
528     /*  0 */ 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xfe,
529     /*  8 */ 0x00, 0x81, 0x81, 0x00, 0x00, 0x81, 0x81, 0x00,
530     /* 16 */ 0x00, 0x81, 0x81, 0x00, 0x00, 0x81, 0x81, 0x00,
531     /* 24 */ 0x00, 0x81, 0x81, 0x00, 0x00, 0x81, 0x81, 0x00,
532     /* 32 */ 0x00, 0x81, 0x81, 0x00, 0x00, 0x81, 0x81, 0x00,
533     /* 40 */ 0x00, 0x81, 0x81, 0x00, 0x00, 0x81, 0x81, 0x00,
534     /* 48 */ 0x00, 0x81, 0x81, 0x00, 0x00, 0x81, 0x81, 0x00,
535     /* 56 */ 0x00, 0x81, 0x81, 0x00, 0x00, 0x81, 0x81, 0x00,
536     /* 64 */ 0x00, 0x81, 0x81, 0x00, 0x00, 0x81, 0x81, 0x00
537   };
538
539
540   /*
541 First move "U+" left and insert "00" after it.
542 */
543   j = 15; /* rows are written bottom to top, so we'll decrement j */
544   for (i = 543 - 8; i > 544 - 24; i--) {
545     offset = 72 * i;
546     image_bytes [offset + 0] = image_bytes [offset + 2];
547     image_bytes [offset + 1] = image_bytes [offset + 3];
548     image_bytes [offset + 2] = image_bytes [offset + 4];
549     image_bytes [offset + 3] = image_bytes [offset + 4] =
550       ~zero_pattern[15 - j--] & 0xFF;
551   }
552
553   /*
554 Now move glyph bitmaps to the right by 8 pixels.
555 */
556   for (i = 0; i < 16; i++) { /* for each glyph row */
557     for (j = 0; j < 16; j++) { /* for each glyph column */
558       /* set offset to lower left-hand byte of next glyph */
559       offset = (32 * 72 * i) + (9 * 72) + (4 * j) + 8;
560       for (k = 0; k < 16; k++) { /* for each glyph row */
561         glyph_row = (image_bytes [offset + 0] « 24) |
562                     (image_bytes [offset + 1] « 16) |
563                     (image_bytes [offset + 2] «  8) |
564                     (image_bytes [offset + 3]);
565         last_pixel = glyph_row & 1;  /* preserve border */
566         glyph_row »= 4;
567         glyph_row &=  0x0FFFFFFE;
568         /* Set left 4 pixels to white and preserve last pixel */
569         glyph_row |=  0xF0000000 | last_pixel;
570         image_bytes [offset + 3] = glyph_row & 0xFF;
571         glyph_row »= 8;
572         image_bytes [offset + 2] = glyph_row & 0xFF;
573         glyph_row »= 8;
574         image_bytes [offset + 1] = glyph_row & 0xFF;
575         glyph_row »= 8;
576         image_bytes [offset + 0] = glyph_row & 0xFF;
577         offset += 72;  /* move up to next row in current glyph */
578       }
579     }
580   }
581
582   /* Replace horizontal grid with unihex2bmp.c grid */
583   for (i = 0; i <= 16; i++) {
584     offset = 32 * 72 * i;
585     for (j = 0; j < 72; j++) {
586       image_bytes [offset + j] = hgrid [j];
587     }
588   }
589
590   return;
591 }
```

## 5.6 src/unicoverage.c File Reference

unicoverage - Show the coverage of Unicode plane scripts for a GNU Unifont hex glyph file

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
Include dependency graph for unicoverage.c:




### Macros

- #define MAXBUF 256
  Maximum input line length - 1.

### Functions

- int main (int argc, char *argv[])
  The main function.
- int nextrange (FILE *coveragefp, int *cstart, int *cend, char *coverstring)
  Get next Unicode range.
- void print_subtotal (FILE *outfp, int print_n, int nglyphs, int cstart, int cend, char *coverstring)
  Print the subtotal for one Unicode script range.

### 5.6.1 Detailed Description

unicoverage - Show the coverage of Unicode plane scripts for a GNU Unifont hex glyph file

Author

Paul Hardy, unifoundry <at> unifoundry.com, 6 January 2008

Copyright



Synopsis: unicoverage [-ifont_file.hex] [-ocoverage_file.txt]

This program requires the file "coverage.dat" to be present in the directory from which it is run.

## 5.6.2 Function Documentation

### 5.6.2.1 main()

```
int main (
            int argc,
            char * argv[] )
```

The main function.

Parameters

| | | |
|---|---|---|
| in | argc | The count of command line arguments. |
| in | argv | Pointer to array of command line arguments. |

Returns

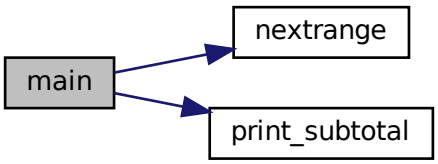
This program exits with status 0.

Definition at line 68 of file unicoverage.c.

```
69 {
70
71   int      print_n=0;       /* print # of glyphs, not percentage   */
72   unsigned i;              /* loop variable                       */
73   unsigned slen;           /* string length of coverage file line */
74   char     inbuf[256];     /* input buffer                        */
75   unsigned thischar;       /* the current character               */
76
77   char *infile="", *outfile="";  /* names of input and output files       */
78   FILE *infp, *outfp;       /* file pointers of input and output files   */
79   FILE *coveragefp;         /* file pointer to coverage.dat file         */
80   int cstart, cend;         /* current coverage start and end code points */
81   char coverstring[MAXBUF];  /* description of current coverage range     */
82   int nglyphs;              /* number of glyphs in this section          */
83   int nextrange();          /* to get next range & name of Unicode glyphs */
84
85   void print_subtotal (FILE *outfp, int print_n, int nglyphs,
86                        int cstart, int cend, char *coverstring);
87
88   if ((coveragefp = fopen ("coverage.dat", "r")) == NULL) {
89     fprintf (stderr, "\nError: data file \"coverage.dat\" not found.\n\n");
90     exit (0);
91   }
92
93   if (argc > 1) {
94     for (i = 1; i < argc; i++) {
95       if (argv[i][0] == '-') {  /* this is an option argument */
96         switch (argv[i][1]) {
97           case 'i':   /* name of input file */
98             infile = &argv[i][2];
99             break;
100          case 'n':   /* print number of glyphs instead of percentage */
101            print_n = 1;
102          case 'o':   /* name of output file */
103            outfile = &argv[i][2];
104            break;
105          default:    /* if unrecognized option, print list and exit */
106            fprintf (stderr, "\nSyntax:\n\n");
107            fprintf (stderr, "    %s -p<Unicode_Page> ", argv[0]);
108            fprintf (stderr, "-i<Input_File> -o<Output_File> -w\n\n");
109            exit (1);
```

```
110             }
111           }
112         }
113       }
114       /*
115 Make sure we can open any I/O files that were specified before
116 doing anything else.
117 */
118       if (strlen (infile) > 0) {
119         if ((infp = fopen (infile, "r")) == NULL) {
120           fprintf (stderr, "Error: can't open %s for input.\n", infile);
121           exit (1);
122         }
123       }
124       else {
125         infp = stdin;
126       }
127       if (strlen (outfile) > 0) {
128         if ((outfp = fopen (outfile, "w")) == NULL) {
129           fprintf (stderr, "Error: can't open %s for output.\n", outfile);
130           exit (1);
131         }
132       }
133       else {
134         outfp = stdout;
135       }
136 
137       /*
138 Print header row.
139 */
140       if (print_n) {
141         fprintf (outfp, "# Glyphs      Range        Script\n");
142         fprintf (outfp, "--------      -----        ------\n");
143       }
144       else {
145         fprintf (outfp, "Covered      Range        Script\n");
146         fprintf (outfp, "-------      -----        ------\n\n");
147       }
148 
149       slen = nextrange (coveragefp, &cstart, &cend, coverstring);
150       nglyphs = 0;
151 
152       /*
153 Read in the glyphs in the file
154 */
155       while (slen != 0 && fgets (inbuf, MAXBUF-1, infp) != NULL) {
156         sscanf (inbuf, "%x", &thischar);
157 
158         /* Read a character beyond end of current script. */
159         while (cend < thischar && slen != 0) {
160           print_subtotal (outfp, print_n, nglyphs, cstart, cend, coverstring);
161 
162           /* start new range total */
163           slen = nextrange (coveragefp, &cstart, &cend, coverstring);
164           nglyphs = 0;
165         }
166         nglyphs++;
167       }
168 
169       print_subtotal (outfp, print_n, nglyphs, cstart, cend, coverstring);
170 
171       exit (0);
172 }
```

Here is the call graph for this function:




#### 5.6.2.2 nextrange()

```
int nextrange (
                FILE * coveragefp,
                int * cstart,
                int * cend,
                char * coverstring )
```

Get next Unicode range.

This function reads the next Unicode script range to count its glyph coverage.

Parameters

| | | |
|---|---|---|
| in | coveragefp | File pointer to Unicode script range data file. |
| in | cstart | Starting code point in current Unicode script range. |
| in | cend | Ending code point in current Unicode script range. |
| out | coverstring | String containing <cstart>-<cend> substring. |

Returns

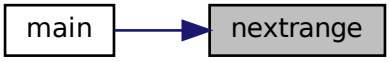
Length of the last string read, or 0 for end of file.

Definition at line 187 of file unicoverage.c.

```
190 {
191   int i;
192   static char inbuf[MAXBUF];
193   int retval;         /* the return value */
194
195   retval = 0;
196
197   do {
198     if (fgets (inbuf, MAXBUF-1, coveragefp) != NULL) {
199       retval = strlen (inbuf);
```

```
200         if ((inbuf[0] >= '0' && inbuf[0] <= '9') ||
201             (inbuf[0] >= 'A' && inbuf[0] <= 'F') ||
202             (inbuf[0] >= 'a' && inbuf[0] <= 'f')) {
203           sscanf (inbuf, "%x-%x", cstart, cend);
204           i = 0;
205           while (inbuf[i] != ' ') i++;  /* find first blank */
206           while (inbuf[i] == ' ') i++;  /* find next non-blank */
207           strncpy (coverstring, &inbuf[i], MAXBUF);
208         }
209         else retval = 0;
210       }
211       else retval = 0;
212     } while (retval == 0 && !feof (coveragefp));
213
214     return (retval);
215 }
```

Here is the caller graph for this function:




#### 5.6.2.3 print_subtotal()

```
void print_subtotal (
                FILE * outfp,
                int print_n,
                int nglyphs,
                int cstart,
                int cend,
                char * coverstring )
```

Print the subtotal for one Unicode script range.

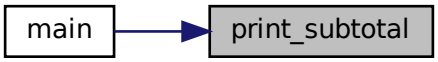
Parameters

| | | |
|---|---|---|
| in | outfp | Pointer to output file. |
| in | print_n | 1 = print number of glyphs, 0 = print percentage. |
| in | nglyphs | Number of glyphs in current range. |
| in | cstart | Starting code point for current range. |
| in | cend | Ending code point for current range. |
| in | coverstring | Character string of "<cstart>-<cend>". |

Definition at line 228 of file unicoverage.c.

```
229                                                 {
230
231     /* print old range total */
232     if (print_n) {  /* Print number of glyphs, not percentage */
233       fprintf (outfp, " %6d ", nglyphs);
234     }
235     else {
236       fprintf (outfp, " %5.1f%%", 100.0*nglyphs/(1+cend-cstart));
237     }
238
239     if (cend < 0x10000)
240       fprintf (outfp, "  U+%04X..U+%04X   %s",
241                cstart, cend, coverstring);
242     else
243       fprintf (outfp, " U+%05X..U+%05X  %s",
244                cstart, cend, coverstring);
245
246     return;
247 }
```

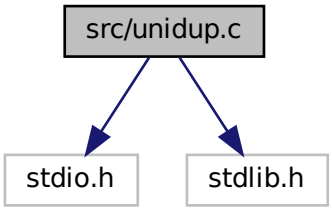
Here is the caller graph for this function:




## 5.7 src/unidup.c File Reference

unidup - Check for duplicate code points in sorted unifont.hex file

#include <stdio.h>
#include <stdlib.h>
Include dependency graph for unidup.c:




### Macros

- #define MAXBUF 256

  Maximum input line length - 1.

## Functions

- int main (int argc, char **argv)
  The main function.

### 5.7.1 Detailed Description

unidup - Check for duplicate code points in sorted unifont.hex file

Author

Paul Hardy, unifoundry <at> unifoundry.com, December 2007

Copyright

Copyright (C) 2007, 2008, 2013 Paul Hardy

This program reads a sorted list of glyphs in Unifont .hex format and prints duplicate code points on stderr if any were detected.

Synopsis: unidup < unifont_file.hex

[Hopefully there won't be any output!]

### 5.7.2 Function Documentation

#### 5.7.2.1 main()

```
int main (
            int argc,
            char ** argv )
```

The main function.

Parameters

| | | |
|---|---|---|
| in | argc | The count of command line arguments. |
| in | argv | Pointer to array of command line arguments. |

Returns

This program exits with status 0.

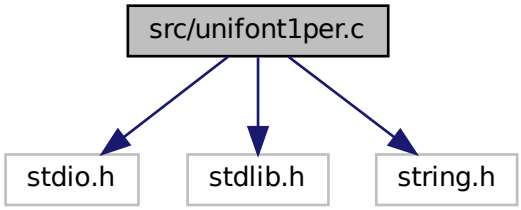
Definition at line 48 of file unidup.c.

```
49 {
50
51    int ix, iy;
52    char inbuf[MAXBUF];
53    char *infile;    /* the input file name */
54    FILE *infilefp; /* file pointer to input file */
55
56    if (argc > 1) {
57       infile = argv[1];
58       if ((infilefp = fopen (infile, "r")) == NULL) {
59          fprintf (stderr, "\nERROR: Can't open file %s\n\n", infile);
60          exit (EXIT_FAILURE);
61       }
62    }
63    else {
64       infilefp = stdin;
65    }
66
67    ix = -1;
68
69    while (fgets (inbuf, MAXBUF-1, infilefp) != NULL) {
70       sscanf (inbuf, "%X", &iy);
71       if (ix == iy) fprintf (stderr, "Duplicate code point: %04X\n", ix);
72       else ix = iy;
73    }
74    exit (0);
75 }
```

## 5.8 src/unifont1per.c File Reference

unifont1per - Read a Unifont .hex file from standard input and produce one glyph per ".bmp" bitmap file as output

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
Include dependency graph for unifont1per.c:




### Macros

- #define MAXSTRING 266
- #define MAXFILENAME 20

## Functions

- int main ()
  The main function.

### 5.8.1 Detailed Description

unifont1per - Read a Unifont .hex file from standard input and produce one glyph per ".bmp" bitmap file as output

Author

Paul Hardy, unifoundry <at> unifoundry.com, December 2016

Copyright



Each glyph is 16 pixels tall, and can be 8, 16, 24, or 32 pixels wide. The width of each output graphic file is determined automatically by the width of each Unifont hex representation.

This program creates files of the form "U+<codepoint>.bmp", 1 per glyph.

Synopsis: unifont1per < unifont.hex

### 5.8.2 Macro Definition Documentation

#### 5.8.2.1 MAXFILENAME

#define MAXFILENAME 20

Maximum size of a filename of the form "U+%06X.bmp".

Definition at line 60 of file unifont1per.c.

#### 5.8.2.2 MAXSTRING

#define MAXSTRING 266

Maximum size of an input line in a Unifont .hex file - 1.

Definition at line 57 of file unifont1per.c.

### 5.8.3 Function Documentation

#### 5.8.3.1 main()

int main ( )

The main function.

Returns

This program exits with status EXIT_SUCCESS.

Definition at line 69 of file unifont1per.c.

```
69        {
70
71    int i; /* loop variable */
72
73    /*
74 Define bitmap header bytes
75 */
76    unsigned char header [62] = {
77       /*
78 Bitmap File Header -- 14 bytes
79 */
80       'B', 'M',       /* Signature          */
81       0x7E, 0, 0, 0,  /* File Size          */
82          0, 0, 0, 0,  /* Reserved           */
83       0x3E, 0, 0, 0,  /* Pixel Array Offset */
84
85       /*
86 Device Independent Bitmap Header -- 40 bytes
87
88 Image Width and Image Height are assigned final values
89 based on the dimensions of each glyph.
90 */
91       0x28,    0,    0,    0,  /* DIB Header Size                */
92       0x10,    0,    0,    0,  /* Image Width = 16 pixels        */
93       0xF0, 0xFF, 0xFF, 0xFF,  /* Image Height = -16 pixels      */
94       0x01,    0,              /* Planes                         */
95       0x01,    0,              /* Bits Per Pixel                 */
96          0,    0,    0,    0,  /* Compression                    */
97       0x40,    0,    0,    0,  /* Image Size                     */
98       0x14, 0x0B,    0,    0,  /* X Pixels Per Meter = 72 dpi    */
99       0x14, 0x0B,    0,    0,  /* Y Pixels Per Meter = 72 dpi    */
100       0x02,    0,    0,    0,  /* Colors In Color Table          */
101          0,    0,    0,    0,  /* Important Colors               */
102
103       /*
104 Color Palette -- 8 bytes
105 */
106       0xFF, 0xFF, 0xFF, 0,  /* White */
107          0,    0,    0, 0   /* Black */
108    };
109
110    char instring[MAXSTRING]; /* input string                      */
111    int  code_point;          /* current Unicode code point        */
112    char glyph[MAXSTRING];    /* bitmap string for this glyph      */
113    int  glyph_height=16;     /* for now, fixed at 16 pixels high  */
114    int  glyph_width;         /* 8, 16, 24, or 32 pixels wide      */
115    char filename[MAXFILENAME];/* name of current output file      */
116    FILE *outfp;              /* file pointer to current output file */
117
118    int string_index;  /* pointer into hexadecimal glyph string */
119    int nextbyte;      /* next set of 8 bits to print out       */
120
121    /* Repeat for each line in the input stream */
122    while (fgets (instring, MAXSTRING - 1, stdin) != NULL) {
123       /* Read next Unifont ASCII hexadecimal format glyph description */
```

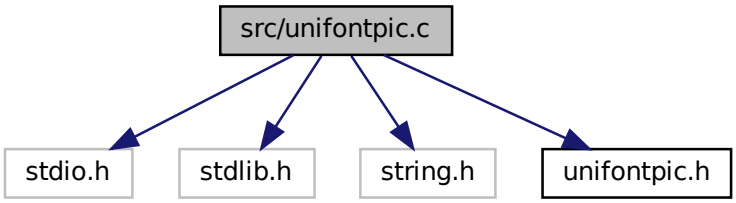
```
124       sscanf (instring, "%X:%s", &code_point, glyph);
125       /* Calculate width of a glyph in pixels; 4 bits per ASCII hex digit */
126       glyph_width = strlen (glyph) / (glyph_height / 4);
127       snprintf (filename, MAXFILENAME, "U+%06X.bmp", code_point);
128       header [18] =  glyph_width;  /* bitmap width */
129       header [22] = -glyph_height; /* negative height --> draw top to bottom */
130       if ((outfp = fopen (filename, "w")) != NULL) {
131         for (i = 0; i < 62; i++) fputc (header[i], outfp);
132         /*
133 Bitmap, with each row padded with zeroes if necessary
134 so each row is four bytes wide.  (Each row must end
135 on a four-byte boundary, and four bytes is the maximum
136 possible row length for up to 32 pixels in a row.)
137 */
138         string_index = 0;
139         for (i = 0; i < glyph_height; i++) {
140           /* Read 2 ASCII hexadecimal digits (1 byte of output pixels) */
141           sscanf (&glyph[string_index], "%2X", &nextbyte);
142           string_index += 2;
143           fputc (nextbyte, outfp);  /* write out the 8 pixels    */
144           if (glyph_width <= 8) {   /* pad row with 3 zero bytes */
145             fputc (0x00, outfp); fputc (0x00, outfp); fputc (0x00, outfp);
146           }
147           else { /* get 8 more pixels */
148             sscanf (&glyph[string_index], "%2X", &nextbyte);
149             string_index += 2;
150             fputc (nextbyte, outfp);  /* write out the 8 pixels    */
151             if (glyph_width <= 16) {  /* pad row with 2 zero bytes */
152               fputc (0x00, outfp); fputc (0x00, outfp);
153             }
154             else { /* get 8 more pixels */
155               sscanf (&glyph[string_index], "%2X", &nextbyte);
156               string_index += 2;
157               fputc (nextbyte, outfp);  /* write out the 8 pixels   */
158               if (glyph_width <= 24) {  /* pad row with 1 zero byte */
159                 fputc (0x00, outfp);
160               }
161               else { /* get 8 more pixels */
162                 sscanf (&glyph[string_index], "%2X", &nextbyte);
163                 string_index += 2;
164                 fputc (nextbyte, outfp); /* write out the 8 pixels */
165               }  /* glyph is 32 pixels wide */
166             }  /* glyph is 24 pixels wide */
167           }  /* glyph is 16 pixels wide */
168         }  /* glyph is 8 pixels wide */
169
170         fclose (outfp);
171       }
172     }
173
174     exit (EXIT_SUCCESS);
175 }
```

## 5.9 src/unifontpic.c File Reference

unifontpic - See the "Big Picture": the entire Unifont in one BMP bitmap

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "unifontpic.h"
```

Include dependency graph for unifontpic.c:




## Macros

- #define HDR_LEN 33

## Functions

- int main (int argc, char **argv)
  The main function.
- void output4 (int thisword)
  Output a 4-byte integer in little-endian order.
- void output2 (int thisword)
  Output a 2-byte integer in little-endian order.
- void gethex (char *instring, int plane_array[0x10000][16], int plane)
  Read a Unifont .hex-format input file from stdin.
- void genlongbmp (int plane_array[0x10000][16], int dpi, int tinynum, int plane)
  Generate the BMP output file in long format.
- void genwidebmp (int plane_array[0x10000][16], int dpi, int tinynum, int plane)
  Generate the BMP output file in wide format.

### 5.9.1 Detailed Description

unifontpic - See the "Big Picture": the entire Unifont in one BMP bitmap

Author

Paul Hardy, 2013

Copyright

## 5.9.2 Macro Definition Documentation

### 5.9.2.1 HDR_LEN

```
#define HDR_LEN 33
```

Define length of header string for top of chart.

Definition at line 67 of file unifontpic.c.

## 5.9.3 Function Documentation

### 5.9.3.1 genlongbmp()

```
void genlongbmp (
            int plane_array[0x10000][16],
            int dpi,
            int tinynum,
            int plane )
```

Generate the BMP output file in long format.

This function generates the BMP output file from a bitmap parameter. This is a long bitmap, 16 glyphs wide by 4,096 glyphs tall.

Parameters

| | | |
|---|---|---|
| in | plane_array | The array of glyph bitmaps for a plane. |
| in | dpi | Dots per inch, for encoding in the BMP output file header. |
| in | tinynum | Whether to generate tiny numbers in wide grid (unused). |
| in | plane | The Unicode plane, 0..17. |

Definition at line 294 of file unifontpic.c.

```
295 {
296
297    char header_string[HDR_LEN]; /* centered header             */
298    char raw_header[HDR_LEN];    /* left-aligned header         */
299    int header[16][16];      /* header row, for chart title */
300    int hdrlen;              /* length of HEADER_STRING     */
301    int startcol;           /* column to start printing header, for centering */
302
303    unsigned leftcol[0x1000][16]; /* code point legend on left side of chart */
304    int d1, d2, d3, d4;           /* digits for filling leftcol[][] legend   */
305    int codept;                  /* current starting code point for legend  */
306    int thisrow;                 /* glyph row currently being rendered      */
307    unsigned toprow[16][16];     /* code point legend on top of chart       */
```

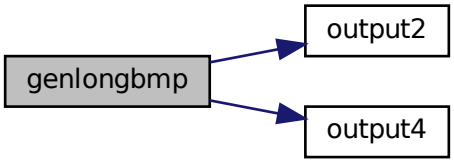
```
308   int digitrow;       /* row we're in (0..4) for the above hexdigit digits */
309
310   /*
311 DataOffset = BMP Header bytes + InfoHeader bytes + ColorTable bytes.
312 */
313   int DataOffset = 14 + 40 + 8; /* fixed size for monochrome BMP */
314   int ImageSize;
315   int FileSize;
316   int Width, Height; /* bitmap image width and height in pixels */
317   int ppm;     /* integer pixels per meter */
318
319   int i, j, k;
320
321   unsigned bytesout;
322
323   void output4(int), output2(int);
324
325   /*
326 Image width and height, in pixels.
327
328 N.B.: Width must be an even multiple of 32 pixels, or 4 bytes.
329 */
330   Width  =   18 * 16;  /* (2 legend +   16 glyphs) * 16 pixels/glyph */
331   Height = 4099 * 16;  /* (1 header + 4096 glyphs) * 16 rows/glyph   */
332
333   ImageSize = Height * (Width / 8); /* in bytes, calculated from pixels */
334
335   FileSize = DataOffset + ImageSize;
336
337   /* convert dots/inch to pixels/meter */
338   if (dpi == 0) dpi = 96;
339   ppm = (int)((double)dpi * 100.0 / 2.54 + 0.5);
340
341   /*
342 Generate the BMP Header
343 */
344   putchar ('B');
345   putchar ('M');
346
347   /*
348 Calculate file size:
349
350 BMP Header + InfoHeader + Color Table + Raster Data
351 */
352   output4 (FileSize);  /* FileSize */
353   output4 (0x0000); /* reserved */
354
355   /* Calculate DataOffset */
356   output4 (DataOffset);
357
358   /*
359 InfoHeader
360 */
361   output4 (40);         /* Size of InfoHeader                    */
362   output4 (Width);      /* Width of bitmap in pixels             */
363   output4 (Height);     /* Height of bitmap in pixels            */
364   output2 (1);          /* Planes (1 plane)                      */
365   output2 (1);          /* BitCount (1 = monochrome)             */
366   output4 (0);          /* Compression (0 = none)                */
367   output4 (ImageSize);  /* ImageSize, in bytes                   */
368   output4 (ppm);        /* XpixelsPerM (96 dpi = 3780 pixels/meter) */
369   output4 (ppm);        /* YpixelsPerM (96 dpi = 3780 pixels/meter) */
370   output4 (2);          /* ColorsUsed (= 2)                      */
371   output4 (2);          /* ColorsImportant (= 2)                 */
372   output4 (0x00000000); /* black (reserved, B, G, R)             */
373   output4 (0x00FFFFFF); /* white (reserved, B, G, R)             */
374
375   /*
376 Create header row bits.
377 */
378   snprintf (raw_header, HDR_LEN, "%s Plane %d", HEADER_STRING, plane);
379   memset ((void *)header, 0, 16 * 16 * sizeof (int)); /* fill with white */
380   memset ((void *)header_string, ' ', 32 * sizeof (char)); /* 32 spaces */
381   header_string[32] = '\0';  /* null-terminated */
382
383   hdrlen = strlen (raw_header);
384   if (hdrlen > 32) hdrlen = 32;       /* only 32 columns to print header */
385   startcol = 16 - ((hdrlen + 1) » 1); /* to center header                */
386   /* center up to 32 chars */
387   memcpy (&header_string[startcol], raw_header, hdrlen);
388
```

```
389   /* Copy each letter's bitmap from the plane_array[][] we constructed. */
390   /* Each glyph must be single-width, to fit two glyphs in 16 pixels */
391   for (j = 0; j < 16; j++) {
392     for (i = 0; i < 16; i++) {
393       header[i][j] =
394         (ascii_bits[header_string[j+j  ] & 0x7F][i] & 0xFF00) |
395         (ascii_bits[header_string[j+j+1] & 0x7F][i] » 8);
396     }
397   }
398
399   /*
400 Create the left column legend.
401 */
402   memset ((void *)leftcol, 0, 4096 * 16 * sizeof (unsigned));
403
404   for (codept = 0x0000; codept < 0x10000; codept += 0x10) {
405     d1 = (codept » 12) & 0xF; /* most significant hex digit */
406     d2 = (codept »  8) & 0xF;
407     d3 = (codept »  4) & 0xF;
408
409     thisrow = codept » 4; /* rows of 16 glyphs */
410
411     /* fill in first and second digits */
412     for (digitrow = 0; digitrow < 5; digitrow++) {
413       leftcol[thisrow][2 + digitrow] =
414         (hexdigit[d1][digitrow] « 10) |
415         (hexdigit[d2][digitrow] «  4);
416     }
417
418     /* fill in third digit */
419     for (digitrow = 0; digitrow < 5; digitrow++) {
420       leftcol[thisrow][9 + digitrow] = hexdigit[d3][digitrow] « 10;
421     }
422     leftcol[thisrow][9 + 4] |= 0xF « 4; /* underscore as 4th digit */
423
424     for (i = 0; i < 15; i ++) {
425       leftcol[thisrow][i] |= 0x00000002;      /* right border */
426     }
427
428     leftcol[thisrow][15] = 0x0000FFFE;        /* bottom border */
429
430     if (d3 == 0xF) {                      /* 256-point boundary */
431       leftcol[thisrow][15] |= 0x00FF0000;  /* longer tic mark */
432     }
433
434     if ((thisrow % 0x40) == 0x3F) {     /* 1024-point boundary */
435       leftcol[thisrow][15] |= 0xFFFF0000; /* longest tic mark */
436     }
437   }
438
439   /*
440 Create the top row legend.
441 */
442   memset ((void *)toprow, 0, 16 * 16 * sizeof (unsigned));
443
444   for (codept = 0x0; codept <= 0xF; codept++) {
445     d1 = (codept » 12) & 0xF; /* most significant hex digit */
446     d2 = (codept »  8) & 0xF;
447     d3 = (codept »  4) & 0xF;
448     d4 =  codept       & 0xF; /* least significant hex digit */
449
450     /* fill in last digit */
451     for (digitrow = 0; digitrow < 5; digitrow++) {
452       toprow[6 + digitrow][codept] = hexdigit[d4][digitrow] « 6;
453     }
454   }
455
456   for (j = 0; j < 16; j++) {
457     /* force bottom pixel row to be white, for separation from glyphs */
458     toprow[15][j] = 0x0000;
459   }
460
461   /* 1 pixel row with left-hand legend line */
462   for (j = 0; j < 16; j++) {
463     toprow[14][j] |= 0xFFFF;
464   }
465
466   /* 14 rows with line on left to fill out this character row */
467   for (i = 13; i >= 0; i--) {
468     for (j = 0; j < 16; j++) {
469       toprow[i][j] |= 0x0001;
```

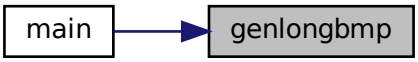
```
470       }
471     }
472
473     /*
474 Now write the raster image.
475
476 XOR each byte with 0xFF because black = 0, white = 1 in BMP.
477 */
478
479     /* Write the glyphs, bottom-up, left-to-right, in rows of 16 (i.e., 0x10) */
480     for (i = 0xFFF0; i >= 0; i -= 0x10) {
481       thisrow = i » 4; /* 16 glyphs per row */
482       for (j = 15; j >= 0; j--) {
483         /* left-hand legend */
484         putchar ((~leftcol[thisrow][j] » 24) & 0xFF);
485         putchar ((~leftcol[thisrow][j] » 16) & 0xFF);
486         putchar ((~leftcol[thisrow][j] »  8) & 0xFF);
487         putchar ( ~leftcol[thisrow][j]       & 0xFF);
488         /* Unifont glyph */
489         for (k = 0; k < 16; k++) {
490           bytesout = ~plane_array[i+k][j] & 0xFFFF;
491           putchar ((bytesout » 8) & 0xFF);
492           putchar ( bytesout      & 0xFF);
493         }
494       }
495     }
496
497     /*
498 Write the top legend.
499 */
500     /* i == 15: bottom pixel row of header is output here */
501     /* left-hand legend: solid black line except for right-most pixel */
502     putchar (0x00);
503     putchar (0x00);
504     putchar (0x00);
505     putchar (0x01);
506     for (j = 0; j < 16; j++) {
507       putchar ((~toprow[15][j] » 8) & 0xFF);
508       putchar ( ~toprow[15][j]      & 0xFF);
509     }
510
511     putchar (0xFF);
512     putchar (0xFF);
513     putchar (0xFF);
514     putchar (0xFC);
515     for (j = 0; j < 16; j++) {
516       putchar ((~toprow[14][j] » 8) & 0xFF);
517       putchar ( ~toprow[14][j]      & 0xFF);
518     }
519
520     for (i = 13; i >= 0; i--) {
521       putchar (0xFF);
522       putchar (0xFF);
523       putchar (0xFF);
524       putchar (0xFD);
525       for (j = 0; j < 16; j++) {
526         putchar ((~toprow[i][j] » 8) & 0xFF);
527         putchar ( ~toprow[i][j]      & 0xFF);
528       }
529     }
530
531     /*
532 Write the header.
533 */
534
535     /* 7 completely white rows */
536     for (i = 7; i >= 0; i--) {
537       for (j = 0; j < 18; j++) {
538         putchar (0xFF);
539         putchar (0xFF);
540       }
541     }
542
543     for (i = 15; i >= 0; i--) {
544       /* left-hand legend */
545       putchar (0xFF);
546       putchar (0xFF);
547       putchar (0xFF);
548       putchar (0xFF);
549       /* header glyph */
550       for (j = 0; j < 16; j++) {
```

```
551         bytesout = ~header[i][j] & 0xFFFF;
552         putchar ((bytesout » 8) & 0xFF);
553         putchar ( bytesout      & 0xFF);
554       }
555     }
556
557     /* 8 completely white rows at very top */
558     for (i = 7; i >= 0; i--) {
559       for (j = 0; j < 18; j++) {
560       putchar (0xFF);
561       putchar (0xFF);
562       }
563     }
564
565     return;
566 }
```

Here is the call graph for this function:




Here is the caller graph for this function:




#### 5.9.3.2 genwidebmp()

```
void genwidebmp (
                int plane_array[0x10000][16],
                int dpi,
                int tinynum,
                int plane )
```

Generate the BMP output file in wide format.

This function generates the BMP output file from a bitmap parameter. This is a wide bitmap, 256 glyphs wide by 256 glyphs tall.

Parameters

| | | |
|---|---|---|
| in | plane_array | The array of glyph bitmaps for a plane. |
| in | dpi | Dots per inch, for encoding in the BMP output file header. |
| in | tinynum | Whether to generate tiny numbers in 256x256 grid. |
| in | plane | The Unicode plane, 0..17. |

Definition at line 581 of file unifontpic.c.

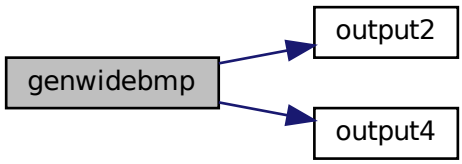
```
582 {
583
584   char header_string[257];
585   char raw_header[HDR_LEN];
586   int header[16][256]; /* header row, for chart title */
587   int hdrlen;          /* length of HEADER_STRING */
588   int startcol;        /* column to start printing header, for centering */
589
590   unsigned leftcol[0x100][16]; /* code point legend on left side of chart */
591   int d1, d2, d3, d4;          /* digits for filling leftcol[][] legend   */
592   int codept;                  /* current starting code point for legend  */
593   int thisrow;                 /* glyph row currently being rendered      */
594   unsigned toprow[32][256];    /* code point legend on top of chart       */
595   int digitrow;       /* row we're in (0..4) for the above hexdigit digits */
596   int hexalpha1, hexalpha2;    /* to convert hex digits to ASCII          */
597
598   /*
599 DataOffset = BMP Header bytes + InfoHeader bytes + ColorTable bytes.
600 */
601   int DataOffset = 14 + 40 + 8; /* fixed size for monochrome BMP */
602   int ImageSize;
603   int FileSize;
604   int Width, Height; /* bitmap image width and height in pixels */
605   int ppm;     /* integer pixels per meter */
606
607   int i, j, k;
608
609   unsigned bytesout;
610
611   void output4(int), output2(int);
612
613   /*
614 Image width and height, in pixels.
615
616 N.B.: Width must be an even multiple of 32 pixels, or 4 bytes.
617 */
618   Width  = 258 * 16;  /* (           2 legend + 256 glyphs) * 16 pixels/glyph */
619   Height = 260 * 16;  /* (2 header + 2 legend + 256 glyphs) * 16 rows/glyph   */
620
621   ImageSize = Height * (Width / 8); /* in bytes, calculated from pixels */
622
623   FileSize = DataOffset + ImageSize;
624
625   /* convert dots/inch to pixels/meter */
626   if (dpi == 0) dpi = 96;
627   ppm = (int)((double)dpi * 100.0 / 2.54 + 0.5);
628
629   /*
630 Generate the BMP Header
631 */
632   putchar ('B');
633   putchar ('M');
634   /*
635 Calculate file size:
636
637 BMP Header + InfoHeader + Color Table + Raster Data
638 */
639   output4 (FileSize);  /* FileSize */
640   output4 (0x0000); /* reserved */
641   /* Calculate DataOffset */
642   output4 (DataOffset);
643
644   /*
645 InfoHeader
646 */
647   output4 (40);        /* Size of InfoHeader                    */
```

```
648     output4 (Width);      /* Width of bitmap in pixels               */
649     output4 (Height);     /* Height of bitmap in pixels              */
650     output2 (1);          /* Planes (1 plane)                     */
651     output2 (1);          /* BitCount (1 = monochrome)               */
652     output4 (0);          /* Compression (0 = none)                  */
653     output4 (ImageSize);  /* ImageSize, in bytes                     */
654     output4 (ppm);        /* XpixelsPerM (96 dpi = 3780 pixels/meter) */
655     output4 (ppm);        /* YpixelsPerM (96 dpi = 3780 pixels/meter) */
656     output4 (2);          /* ColorsUsed (= 2)                     */
657     output4 (2);          /* ColorsImportant (= 2)                   */
658     output4 (0x00000000); /* black (reserved, B, G, R)               */
659     output4 (0x00FFFFFF); /* white (reserved, B, G, R)               */
660
661     /*
662   Create header row bits.
663   */
664     snprintf (raw_header, HDR_LEN, "%s Plane %d", HEADER_STRING, plane);
665     memset ((void *)header, 0, 256 * 16 * sizeof (int)); /* fill with white */
666     memset ((void *)header_string, ' ', 256 * sizeof (char)); /* 256 spaces */
667     header_string[256] = '\0';  /* null-terminated */
668
669     hdrlen = strlen (raw_header);
670     /* Wide bitmap can print 256 columns, but limit to 32 columns for long bitmap. */
671     if (hdrlen > 32) hdrlen = 32;
672     startcol = 127 - ((hdrlen - 1) » 1);  /* to center header */
673     /* center up to 32 chars */
674     memcpy (&header_string[startcol], raw_header, hdrlen);
675
676     /* Copy each letter's bitmap from the plane_array[][] we constructed. */
677     for (j = 0; j < 256; j++) {
678       for (i = 0; i < 16; i++) {
679         header[i][j] = ascii_bits[header_string[j] & 0x7F][i];
680       }
681     }
682
683     /*
684   Create the left column legend.
685   */
686     memset ((void *)leftcol, 0, 256 * 16 * sizeof (unsigned));
687
688     for (codept = 0x0000; codept < 0x10000; codept += 0x100) {
689       d1 = (codept » 12) & 0xF; /* most significant hex digit */
690       d2 = (codept »  8) & 0xF;
691
692       thisrow = codept » 8; /* rows of 256 glyphs */
693
694       /* fill in first and second digits */
695
696       if (tinynum) { /* use 4x5 pixel glyphs */
697         for (digitrow = 0; digitrow < 5; digitrow++) {
698           leftcol[thisrow][6 + digitrow] =
699             (hexdigit[d1][digitrow] « 10) |
700             (hexdigit[d2][digitrow] «  4);
701         }
702       }
703       else { /* bigger numbers -- use glyphs from Unifont itself */
704         /* convert hexadecimal digits to ASCII equivalent */
705         hexalpha1 = d1 < 0xA ? '0' + d1 : 'A' + d1 - 0xA;
706         hexalpha2 = d2 < 0xA ? '0' + d2 : 'A' + d2 - 0xA;
707
708         for (i = 0 ; i < 16; i++) {
709           leftcol[thisrow][i] =
710             (ascii_bits[hexalpha1][i] « 2) |
711             (ascii_bits[hexalpha2][i] » 6);
712         }
713       }
714
715       for (i = 0; i < 15; i ++) {
716         leftcol[thisrow][i] |= 0x00000002;      /* right border */
717       }
718
719       leftcol[thisrow][15] = 0x0000FFFE;        /* bottom border */
720
721       if (d2 == 0xF) {                      /* 4096-point boundary */
722         leftcol[thisrow][15] |= 0x00FF0000;  /* longer tic mark */
723       }
724
725       if ((thisrow % 0x40) == 0x3F) {     /* 16,384-point boundary */
726         leftcol[thisrow][15] |= 0xFFFF0000; /* longest tic mark */
727       }
728     }
```

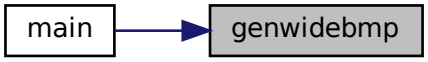
```
729
730   /*
731 Create the top row legend.
732 */
733   memset ((void *)toprow, 0, 32 * 256 * sizeof (unsigned));
734
735   for (codept = 0x00; codept <= 0xFF; codept++) {
736     d3 = (codept »  4) & 0xF;
737     d4 =  codept       & 0xF; /* least significant hex digit */
738
739     if (tinynum) {
740       for (digitrow = 0; digitrow < 5; digitrow++) {
741         toprow[16 + 6 + digitrow][codept] =
742           (hexdigit[d3][digitrow] « 10) |
743           (hexdigit[d4][digitrow] «  4);
744       }
745     }
746     else {
747       /* convert hexadecimal digits to ASCII equivalent */
748       hexalpha1 = d3 < 0xA ? '0' + d3 : 'A' + d3 - 0xA;
749       hexalpha2 = d4 < 0xA ? '0' + d4 : 'A' + d4 - 0xA;
750       for (i = 0 ; i < 16; i++) {
751         toprow[14 + i][codept] =
752           (ascii_bits[hexalpha1][i]     ) |
753           (ascii_bits[hexalpha2][i] » 7);
754       }
755     }
756   }
757
758   for (j = 0; j < 256; j++) {
759     /* force bottom pixel row to be white, for separation from glyphs */
760     toprow[16 + 15][j] = 0x0000;
761   }
762
763   /* 1 pixel row with left-hand legend line */
764   for (j = 0; j < 256; j++) {
765     toprow[16 + 14][j] |= 0xFFFF;
766   }
767
768   /* 14 rows with line on left to fill out this character row */
769   for (i = 13; i >= 0; i--) {
770     for (j = 0; j < 256; j++) {
771       toprow[16 + i][j] |= 0x0001;
772     }
773   }
774
775   /* Form the longer tic marks in top legend */
776   for (i = 8; i < 16; i++) {
777     for (j = 0x0F; j < 0x100; j += 0x10) {
778       toprow[i][j] |= 0x0001;
779     }
780   }
781
782   /*
783 Now write the raster image.
784
785 XOR each byte with 0xFF because black = 0, white = 1 in BMP.
786 */
787
788   /* Write the glyphs, bottom-up, left-to-right, in rows of 16 (i.e., 0x10) */
789   for (i = 0xFF00; i >= 0; i -= 0x100) {
790     thisrow = i » 8; /* 256 glyphs per row */
791     for (j = 15; j >= 0; j--) {
792       /* left-hand legend */
793       putchar ((~leftcol[thisrow][j] » 24) & 0xFF);
794       putchar ((~leftcol[thisrow][j] » 16) & 0xFF);
795       putchar ((~leftcol[thisrow][j] »  8) & 0xFF);
796       putchar ( ~leftcol[thisrow][j]       & 0xFF);
797       /* Unifont glyph */
798       for (k = 0x00; k < 0x100; k++) {
799         bytesout = ~plane_array[i+k][j] & 0xFFFF;
800         putchar ((bytesout » 8) & 0xFF);
801         putchar ( bytesout      & 0xFF);
802       }
803     }
804   }
805
806   /*
807 Write the top legend.
808 */
809   /* i == 15: bottom pixel row of header is output here */
```

```
810     /* left-hand legend: solid black line except for right-most pixel */
811     putchar (0x00);
812     putchar (0x00);
813     putchar (0x00);
814     putchar (0x01);
815     for (j = 0; j < 256; j++) {
816        putchar ((~toprow[16 + 15][j] » 8) & 0xFF);
817        putchar ( ~toprow[16 + 15][j]      & 0xFF);
818     }
819
820     putchar (0xFF);
821     putchar (0xFF);
822     putchar (0xFF);
823     putchar (0xFC);
824     for (j = 0; j < 256; j++) {
825        putchar ((~toprow[16 + 14][j] » 8) & 0xFF);
826        putchar ( ~toprow[16 + 14][j]      & 0xFF);
827     }
828
829     for (i = 16 + 13; i >= 0; i--) {
830        if (i >= 8) { /* make vertical stroke on right */
831           putchar (0xFF);
832           putchar (0xFF);
833           putchar (0xFF);
834           putchar (0xFD);
835        }
836        else { /* all white */
837           putchar (0xFF);
838           putchar (0xFF);
839           putchar (0xFF);
840           putchar (0xFF);
841        }
842        for (j = 0; j < 256; j++) {
843           putchar ((~toprow[i][j] » 8) & 0xFF);
844           putchar ( ~toprow[i][j]      & 0xFF);
845        }
846     }
847
848     /*
849  Write the header.
850  */
851
852     /* 8 completely white rows */
853     for (i = 7; i >= 0; i--) {
854        for (j = 0; j < 258; j++) {
855           putchar (0xFF);
856           putchar (0xFF);
857        }
858     }
859
860     for (i = 15; i >= 0; i--) {
861        /* left-hand legend */
862        putchar (0xFF);
863        putchar (0xFF);
864        putchar (0xFF);
865        putchar (0xFF);
866        /* header glyph */
867        for (j = 0; j < 256; j++) {
868           bytesout = ~header[i][j] & 0xFFFF;
869           putchar ((bytesout » 8) & 0xFF);
870           putchar ( bytesout      & 0xFF);
871        }
872     }
873
874     /* 8 completely white rows at very top */
875     for (i = 7; i >= 0; i--) {
876        for (j = 0; j < 258; j++) {
877        putchar (0xFF);
878        putchar (0xFF);
879        }
880     }
881
882     return;
883  }
```

Here is the call graph for this function:




Here is the caller graph for this function:




#### 5.9.3.3 gethex()

```
void gethex (
            char * instring,
            int plane_array[0x10000][16],
            int plane )
```

Read a Unifont .hex-format input file from stdin.

Each glyph can be 2, 4, 6, or 8 ASCII hexadecimal digits wide. Glyph height is fixed at 16 pixels.

Parameters

| | | |
|---|---|---|
| in | instring | One line from a Unifont .hex-format file. |
| in,out | plane_array | Bitmap for this plane, one bitmap row per element. |
| in | plane | The Unicode plane, 0..17. |

Definition at line 215 of file unifontpic.c.
216 {

```
217   char *bitstring;  /* pointer into instring for glyph bitmap */
218   int i;       /* loop variable                              */
219   int codept;  /* the Unicode code point of the current glyph */
220   int glyph_plane; /* Unicode plane of current glyph          */
221   int ndigits; /* number of ASCII hexadecimal digits in glyph */
222   int bytespl; /* bytes per line of pixels in a glyph        */
223   int temprow; /* 1 row of a quadruple-width glyph           */
224   int newrow;  /* 1 row of double-width output pixels        */
225   unsigned bitmask; /* to mask off 2 bits of long width glyph */
226
227   /*
228 Read each input line and place its glyph into the bit array.
229 */
230   sscanf (instring, "%X", &codept);
231   glyph_plane = codept » 16;
232   if (glyph_plane == plane) {
233     codept &= 0xFFFF;  /* array index will only have 16 bit address */
234     /* find the colon separator */
235     for (i = 0; (i < 9) && (instring[i] != ':'); i++);
236     i++; /* position past it */
237     bitstring = &instring[i];
238     ndigits = strlen (bitstring);
239     /* don't count '\n' at end of line if present */
240     if (bitstring[ndigits - 1] == '\n') ndigits--;
241     bytespl = ndigits » 5;  /* 16 rows per line, 2 digits per byte */
242
243     if (bytespl >= 1 && bytespl <= 4) {
244       for (i = 0; i < 16; i++) { /* 16 rows per glyph */
245         /* Read correct number of hexadecimal digits given glyph width */
246         switch (bytespl) {
247           case 1: sscanf (bitstring, "%2X", &temprow);
248                   bitstring += 2;
249                   temprow «= 8; /* left-justify single-width glyph */
250                   break;
251           case 2: sscanf (bitstring, "%4X", &temprow);
252                   bitstring += 4;
253                   break;
254           /* cases 3 and 4 widths will be compressed by 50% (see below) */
255           case 3: sscanf (bitstring, "%6X", &temprow);
256                   bitstring += 6;
257                   temprow «= 8; /* left-justify */
258                   break;
259           case 4: sscanf (bitstring, "%8X", &temprow);
260                   bitstring += 8;
261                   break;
262         }  /* switch on number of bytes per row */
263         /* compress glyph width by 50% if greater than double-width */
264         if (bytespl > 2) {
265           newrow = 0x0000;
266           /* mask off 2 bits at a time to convert each pair to 1 bit out */
267           for (bitmask = 0xC0000000; bitmask != 0; bitmask »= 2) {
268             newrow «= 1;
269             if ((temprow & bitmask) != 0) newrow |= 1;
270           }
271           temprow = newrow;
272         }  /* done conditioning glyphs beyond double-width */
273         plane_array[codept][i] = temprow;  /* store glyph bitmap for output */
274       }  /* for each row */
275     }  /* if 1 to 4 bytes per row/line */
276   }  /* if this is the plane we are seeking */
277
278   return;
279 }
```

Here is the caller graph for this function:

#### 5.9.3.4 main()

```
int main (
                int argc,
                char ** argv )
```

The main function.

Parameters

| | | |
|---|---|---|
| in | argc | The count of command line arguments. |
| in | argv | Pointer to array of command line arguments. |

Returns

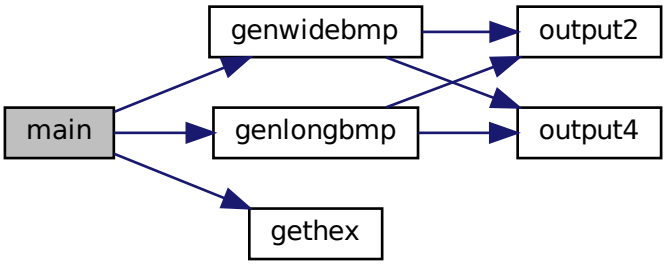
This program exits with status EXIT_SUCCESS.

Definition at line 87 of file unifontpic.c.

```
88 {
89   /* Input line buffer */
90   char instring[MAXSTRING];
91
92   /* long and dpi are set from command-line options */
93   int wide=1; /* =1 for a 256x256 grid, =0 for a 16x4096 grid */
94   int dpi=96; /* change for 256x256 grid to fit paper if desired */
95   int tinynum=0; /* whether to use tiny labels for 256x256 grid */
96
97   int i, j; /* loop variables */
98
99   int plane=0;      /* Unicode plane, 0..17; Plane 0 is default */
100   /* 16 pixel rows for each of 65,536 glyphs in a Unicode plane */
101   int plane_array[0x10000][16];
102
103   void gethex();
104   void genlongbmp();
105   void genwidebmp();
106
107   if (argc > 1) {
108     for (i = 1; i < argc; i++) {
109       if (strncmp (argv[i],"-l",2) == 0) { /* long display */
110         wide = 0;
111       }
112       else if (strncmp (argv[i],"-d",2) == 0) {
113         dpi = atoi (&argv[i][2]); /* dots/inch specified on command line */
114       }
115       else if (strncmp (argv[i],"-t",2) == 0) {
116         tinynum = 1;
117       }
118       else if (strncmp (argv[i],"-P",2) == 0) {
119         /* Get Unicode plane */
120         for (j = 2; argv[i][j] != '\0'; j++) {
121           if (argv[i][j] < '0' || argv[i][j] > '9') {
122             fprintf (stderr,
123                      "ERROR: Specify Unicode plane as decimal number.\n\n");
124             exit (EXIT_FAILURE);
125           }
126         }
127         plane = atoi (&argv[i][2]); /* Unicode plane, 0..17 */
128         if (plane < 0 || plane > 17) {
129           fprintf (stderr,
130                    "ERROR: Plane out of Unicode range [0,17].\n\n");
131           exit (EXIT_FAILURE);
132         }
```

```
133         }
134       }
135     }
136
137
138     /*
139 Initialize the ASCII bitmap array for chart titles
140 */
141     for (i = 0; i < 128; i++) {
142       gethex (ascii_hex[i], plane_array, 0); /* convert Unifont hexadecimal string to bitmap */
143       for (j = 0; j < 16; j++) ascii_bits[i][j] = plane_array[i][j];
144     }
145
146
147     /*
148 Read in the Unifont hex file to render from standard input
149 */
150     memset ((void *)plane_array, 0, 0x10000 * 16 * sizeof (int));
151     while (fgets (instring, MAXSTRING, stdin) != NULL) {
152       gethex (instring, plane_array, plane); /* read .hex input file and fill plane_array with glyph data */
153     }  /* while not EOF */
154
155
156     /*
157 Write plane_array glyph data to BMP file as wide or long bitmap.
158 */
159     if (wide) {
160       genwidebmp (plane_array, dpi, tinynum, plane);
161     }
162     else {
163       genlongbmp (plane_array, dpi, tinynum, plane);
164     }
165
166     exit (EXIT_SUCCESS);
167 }
```

Here is the call graph for this function:




#### 5.9.3.5 output2()

```
void output2 (
                int thisword )
```

Output a 2-byte integer in little-endian order.

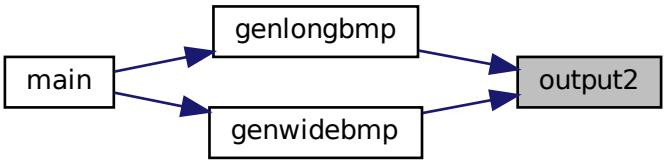
Parameters

| | | |
|---|---|---|
| in | thisword | The 2-byte integer to output as binary data. |

Definition at line 194 of file unifontpic.c.

```
195 {
196
197     putchar ( thisword       & 0xFF);
198     putchar ((thisword » 8) & 0xFF);
199
200     return;
201 }
```

Here is the caller graph for this function:




### 5.9.3.6 output4()

```
void output4 (
                int thisword )
```

Output a 4-byte integer in little-endian order.

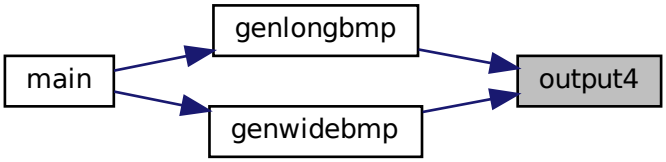
Parameters

| | | |
|---|---|---|
| in | thisword | The 4-byte integer to output as binary data. |

Definition at line 176 of file unifontpic.c.

```
177 {
178
179     putchar ( thisword        & 0xFF);
180     putchar ((thisword »  8) & 0xFF);
181     putchar ((thisword » 16) & 0xFF);
182     putchar ((thisword » 24) & 0xFF);
183
184     return;
185 }
```

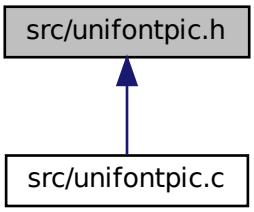
Here is the caller graph for this function:




## 5.10 src/unifontpic.h File Reference

unifontpic.h - Header file for unifontpic.c

This graph shows which files directly or indirectly include this file:




### Macros

- #define MAXSTRING 256
  Maximum input string allowed.
- #define HEADER_STRING "GNU Unifont 15.0.05"
  To be printed as chart title.

### Variables

- const char * ascii_hex [128]
  Array of Unifont ASCII glyphs for chart row & column headings.
- int ascii_bits [128][16]
  Array to hold ASCII bitmaps for chart title.
- char hexdigit [16][5]
  Array of 4x5 hexadecimal digits for legend.

### 5.10.1 Detailed Description

unifontpic.h - Header file for unifontpic.c

Author

Paul Hardy, July 2017

Copyright



### 5.10.2 Variable Documentation

#### 5.10.2.1 ascii_bits

int ascii_bits[128][16]

Array to hold ASCII bitmaps for chart title.

This array will be created from the strings in ascii_hex[] above.

Definition at line 177 of file unifontpic.h.

#### 5.10.2.2 ascii_hex

const char* ascii_hex[128]

Array of Unifont ASCII glyphs for chart row & column headings.

Define the array of Unifont ASCII glyphs, code points 0 through 127. This allows using unifontpic to print charts of glyphs above Unicode Plane 0. These were copied from font/plane00/unifont-base.hex, plus U+0020 (ASCII space character).

Definition at line 40 of file unifontpic.h.

#### 5.10.2.3 hexdigit

char hexdigit[16][5]

Initial value:
```
= {
   {0x6,0x9,0x9,0x9,0x6},
   {0x2,0x6,0x2,0x2,0x7},
   {0xF,0x1,0xF,0x8,0xF},
   {0xE,0x1,0x7,0x1,0xE},
   {0x9,0x9,0xF,0x1,0x1},
   {0xF,0x8,0xF,0x1,0xF},
   {0x6,0x8,0xE,0x9,0x6},
   {0xF,0x1,0x2,0x4,0x4},
   {0x6,0x9,0x6,0x9,0x6},
   {0x6,0x9,0x7,0x1,0x6},
   {0xF,0x9,0xF,0x9,0x9},
   {0xE,0x9,0xE,0x9,0xE},
   {0x7,0x8,0x8,0x8,0x7},
   {0xE,0x9,0x9,0x9,0xE},
   {0xF,0x8,0xE,0x8,0xF},
   {0xF,0x8,0xE,0x8,0x8}
}
```

Array of 4x5 hexadecimal digits for legend.

hexdigit contains 4x5 pixel arrays of tiny digits for the legend. See unihexgen.c for a more detailed description in the comments.

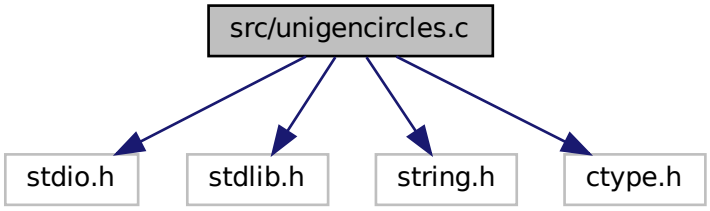
Definition at line 186 of file unifontpic.h.

## 5.11 src/unigencircles.c File Reference

unigencircles - Superimpose dashed combining circles on combining glyphs

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
```

Include dependency graph for unigencircles.c:



## Macros

- #define MAXSTRING 256
  Maximum input line length - 1.

## Functions

- int main (int argc, char **argv)
  The main function.
- void add_single_circle (char *glyphstring)
  Superimpose a single-width dashed combining circle on a glyph bitmap.
- void add_double_circle (char *glyphstring, int offset)
  Superimpose a double-width dashed combining circle on a glyph bitmap.

### 5.11.1 Detailed Description

unigencircles - Superimpose dashed combining circles on combining glyphs

Author

Paul Hardy

Copyright



### 5.11.2 Function Documentation

#### 5.11.2.1 add_double_circle()

```
void add_double_circle (
            char * glyphstring,
            int offset )
```

Superimpose a double-width dashed combining circle on a glyph bitmap.

Parameters

| | | |
|---|---|---|
| in,out | glyphstring | A double-width glyph, 16x16 pixels. |

Definition at line 221 of file unigencircles.c.

```
222 {
223
224   char newstring[256];
225   /* Circle hex string pattern is "00000008000024004200240000000000" */
226
227   /* For double diacritical glyphs (offset = -8) */
228   /* Combining circle is left-justified.          */
229   char circle08[64]={0x0,0x0,0x0,0x0,  /* row  1 */
230                      0x0,0x0,0x0,0x0,  /* row  2 */
231                      0x0,0x0,0x0,0x0,  /* row  3 */
232                      0x0,0x0,0x0,0x0,  /* row  4 */
233                      0x0,0x0,0x0,0x0,  /* row  5 */
234                      0x0,0x0,0x0,0x0,  /* row  6 */
235                      0x2,0x4,0x0,0x0,  /* row  7 */
236                      0x0,0x0,0x0,0x0,  /* row  8 */
237                      0x4,0x2,0x0,0x0,  /* row  9 */
238                      0x0,0x0,0x0,0x0,  /* row 10 */
239                      0x2,0x4,0x0,0x0,  /* row 11 */
240                      0x0,0x0,0x0,0x0,  /* row 12 */
241                      0x0,0x0,0x0,0x0,  /* row 13 */
242                      0x0,0x0,0x0,0x0,  /* row 14 */
243                      0x0,0x0,0x0,0x0,  /* row 15 */
244                      0x0,0x0,0x0,0x0}; /* row 16 */
245
246   /* For all other combining glyphs (offset = -16) */
247   /* Combining circle is centered in 16 columns.   */
248   char circle16[64]={0x0,0x0,0x0,0x0,  /* row  1 */
249                      0x0,0x0,0x0,0x0,  /* row  2 */
250                      0x0,0x0,0x0,0x0,  /* row  3 */
251                      0x0,0x0,0x0,0x0,  /* row  4 */
252                      0x0,0x0,0x0,0x0,  /* row  5 */
253                      0x0,0x0,0x0,0x0,  /* row  6 */
254                      0x0,0x2,0x4,0x0,  /* row  7 */
255                      0x0,0x0,0x0,0x0,  /* row  8 */
256                      0x0,0x4,0x2,0x0,  /* row  9 */
257                      0x0,0x0,0x0,0x0,  /* row 10 */
258                      0x0,0x2,0x4,0x0,  /* row 11 */
259                      0x0,0x0,0x0,0x0,  /* row 12 */
260                      0x0,0x0,0x0,0x0,  /* row 13 */
261                      0x0,0x0,0x0,0x0,  /* row 14 */
262                      0x0,0x0,0x0,0x0,  /* row 15 */
263                      0x0,0x0,0x0,0x0}; /* row 16 */
264
265   char *circle; /* points into circle16 or circle08 */
266
267   int digit1, digit2; /* corresponding digits in each string */
268
269   int i; /* index variables */
270
271
272   /*
273 Determine if combining circle is left-justified (offset = -8)
274 or centered (offset = -16).
275 */
276   circle = (offset >= -8) ? circle08 : circle16;
277
278   /* for each character position, OR the corresponding circle glyph value */
279   for (i = 0; i < 64; i++) {
280     glyphstring[i] = toupper (glyphstring[i]);
281
282     /* Convert ASCII character to a hexadecimal integer */
283     digit1 = (glyphstring[i] <= '9') ?
284              (glyphstring[i] - '0') : (glyphstring[i] - 'A' + 0xA);
285
286     /* Superimpose dashed circle */
287     digit2 = digit1 | circle[i];
288
289     /* Convert hexadecimal integer to an ASCII character */
290     newstring[i] = (digit2 <= 9) ?
291                    ('0' + digit2) : ('A' + digit2 - 0xA);
292   }
293
294   /* Terminate string for output */
295   newstring[i++] = '\n';
296   newstring[i++] = '\0';
297
298   memcpy (glyphstring, newstring, i);
299
300   return;
301 }
```

Here is the caller graph for this function:


#### 5.11.2.2 add_single_circle()

```
void add_single_circle (
                char * glyphstring )
```

Superimpose a single-width dashed combining circle on a glyph bitmap.

Parameters

| in,out | glyphstring | A single-width glyph, 8x16 pixels. |
|---|---|---|

Definition at line 163 of file unigencircles.c.

```
164 {
165
166   char newstring[256];
167   /* Circle hex string pattern is "00000008000024004200240000000000" */
168   char circle[32]={0x0,0x0,  /* row  1 */
169                    0x0,0x0,  /* row  2 */
170                    0x0,0x0,  /* row  3 */
171                    0x0,0x0,  /* row  4 */
172                    0x0,0x0,  /* row  5 */
173                    0x0,0x0,  /* row  6 */
174                    0x2,0x4,  /* row  7 */
175                    0x0,0x0,  /* row  8 */
176                    0x4,0x2,  /* row  9 */
177                    0x0,0x0,  /* row 10 */
178                    0x2,0x4,  /* row 11 */
179                    0x0,0x0,  /* row 12 */
180                    0x0,0x0,  /* row 13 */
181                    0x0,0x0,  /* row 14 */
182                    0x0,0x0,  /* row 15 */
183                    0x0,0x0}; /* row 16 */
184
185   int digit1, digit2; /* corresponding digits in each string */
186
187   int i; /* index variables */
188
189   /* for each character position, OR the corresponding circle glyph value */
190   for (i = 0; i < 32; i++) {
191     glyphstring[i] = toupper (glyphstring[i]);
192
193     /* Convert ASCII character to a hexadecimal integer */
194     digit1 = (glyphstring[i] <= '9') ?
195           (glyphstring[i] - '0') : (glyphstring[i] - 'A' + 0xA);
196
197     /* Superimpose dashed circle */
198     digit2 = digit1 | circle[i];
```

```
199
200       /* Convert hexadecimal integer to an ASCII character */
201       newstring[i] = (digit2 <= 9) ?
202                      ('0' + digit2) : ('A' + digit2 - 0xA);
203     }
204
205     /* Terminate string for output */
206     newstring[i++] = '\n';
207     newstring[i++] = '\0';
208
209     memcpy (glyphstring, newstring, i);
210
211     return;
212 }
```

Here is the caller graph for this function:


### 5.11.2.3 main()

```
int main (
                int argc,
                char ** argv )
```

The main function.

Parameters

| | | |
|---|---|---|
| in | argc | The count of command line arguments. |
| in | argv | Pointer to array of command line arguments. |

Returns

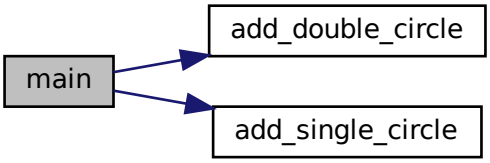
This program exits with status EXIT_SUCCESS.

Definition at line 73 of file unigencircles.c.

```
74 {
75
76     char teststring[MAXSTRING];  /* current input line                    */
77     int  loc;                    /* Unicode code point of current input line */
78     int  offset;                 /* offset value of a combining character    */
79     char *gstart;                  /* glyph start, pointing into teststring    */
80
81     char combining[0x110000];    /* 1 --> combining glyph; 0 --> non-combining */
82     char x_offset [0x110000];    /* second value in *combining.txt files       */
```

```
83
84    void add_single_circle(char *);       /* add a single-width dashed circle */
85    void add_double_circle(char *, int); /* add a double-width dashed circle */
86
87    FILE *infilefp;
88
89    /*
90 if (argc != 3) {
91 fprintf (stderr,
92 "\n\nUsage: %s combining.txt nonprinting.hex < unifont.hex > unifontfull.hex\n\n");
93 exit (EXIT_FAILURE);
94 }
95 */
96
97    /*
98 Read the combining characters list.
99 */
100    /* Start with no combining code points flagged */
101    memset (combining, 0, 0x110000 * sizeof (char));
102    memset (x_offset , 0, 0x110000 * sizeof (char));
103
104    if ((infilefp = fopen (argv[1],"r")) == NULL) {
105       fprintf (stderr,"ERROR - combining characters file %s not found.\n\n",
106             argv[1]);
107       exit (EXIT_FAILURE);
108    }
109
110    /* Flag list of combining characters to add a dashed circle. */
111    while (fscanf (infilefp, "%X:%d", &loc, &offset) != EOF) {
112       /*
113 U+01107F and U+01D1A0 are not defined as combining characters
114 in Unicode; they were added in a combining.txt file as the
115 only way to make them look acceptable in proximity to other
116 glyphs in their script.
117 */
118       if (loc != 0x01107F && loc != 0x01D1A0) {
119          combining[loc] = 1;
120          x_offset [loc] = offset;
121       }
122    }
123    fclose (infilefp); /* all done reading combining.txt */
124
125    /* Now read the non-printing glyphs; they never have dashed circles */
126    if ((infilefp = fopen (argv[2],"r")) == NULL) {
127       fprintf (stderr,"ERROR - nonprinting characters file %s not found.\n\n",
128             argv[1]);
129       exit (EXIT_FAILURE);
130    }
131
132    /* Reset list of nonprinting characters to avoid adding a dashed circle. */
133    while (fscanf (infilefp, "%X:%*s", &loc) != EOF) combining[loc] = 0;
134
135    fclose (infilefp); /* all done reading nonprinting.hex */
136
137    /*
138 Read the hex glyphs.
139 */
140    teststring[MAXSTRING - 1] = '\0';   /* so there's no chance we leave array  */
141    while (fgets (teststring, MAXSTRING-1, stdin) != NULL) {
142       sscanf (teststring, "%X", &loc);     /* loc == the Uniocde code point    */
143       gstart = strchr (teststring,':') + 1; /* start of glyph bitmap            */
144       if (combining[loc]) {                /* if a combining character         */
145          if (strlen (gstart) < 35)
146             add_single_circle (gstart);                /* single-width */
147          else
148             add_double_circle (gstart, x_offset[loc]);  /* double-width */
149       }
150       printf ("%s", teststring); /* output the new character .hex string */
151    }
152
153    exit (EXIT_SUCCESS);
154 }
```

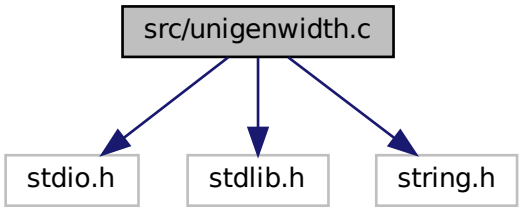
Here is the call graph for this function:




## 5.12 src/unigenwidth.c File Reference

unigenwidth - IEEE 1003.1-2008 setup to calculate wchar_t string widths

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
Include dependency graph for unigenwidth.c:




### Macros

- #define MAXSTRING 256
  Maximum input line length - 1.
- #define PIKTO_START 0x0F0E70
  Start of Pikto code point range.
- #define PIKTO_END 0x0F11EF
  End of Pikto code point range.
- #define PIKTO_SIZE (PIKTO_END - PIKTO_START + 1)

## Functions

- int main (int argc, char **argv)
  The main function.

### 5.12.1 Detailed Description

unigenwidth - IEEE 1003.1-2008 setup to calculate wchar_t string widths

Author

Paul Hardy.

Copyright



All glyphs are treated as 16 pixels high, and can be 8, 16, 24, or 32 pixels wide (resulting in widths of 1, 2, 3, or 4, respectively).

### 5.12.2 Macro Definition Documentation

#### 5.12.2.1 PIKTO_SIZE

#define PIKTO_SIZE (PIKTO_END - PIKTO_START + 1)

Number of code points in Pikto range.

Definition at line 52 of file unigenwidth.c.

### 5.12.3 Function Documentation

#### 5.12.3.1 main()

int main (
int argc,
char ** argv )

The main function.

Parameters

| | | |
|---|---|---|
| in | argc | The count of command line arguments. |
| in | argv | Pointer to array of command line arguments. |

Returns

This program exits with status EXIT_SUCCESS.

Definition at line 63 of file unigenwidth.c.

```
64 {
65
66   int i; /* loop variable */
67
68   char teststring[MAXSTRING];
69   int  loc;
70   char *gstart;
71
72   char glyph_width[0x20000];
73   char pikto_width[PIKTO_SIZE];
74
75   FILE *infilefp;
76
77   if (argc != 3) {
78     fprintf (stderr, "\n\nUsage: %s <unifont.hex> <combining.txt>\n\n", argv[0]);
79     exit (EXIT_FAILURE);
80   }
81
82   /*
83 Read the collection of hex glyphs.
84 */
85   if ((infilefp = fopen (argv[1],"r")) == NULL) {
86     fprintf (stderr,"ERROR - hex input file %s not found.\n\n", argv[1]);
87     exit (EXIT_FAILURE);
88   }
89
90   /* Flag glyph as non-existent until found. */
91   memset (glyph_width, -1, 0x20000 * sizeof (char));
92   memset (pikto_width, -1, (PIKTO_SIZE) * sizeof (char));
93
94   teststring[MAXSTRING-1] = '\0';
95   while (fgets (teststring, MAXSTRING-1, infilefp) != NULL) {
96     sscanf (teststring, "%X:%*s", &loc);
97     if (loc < 0x20000) {
98       gstart = strchr (teststring,':') + 1;
99       /*
100 16 rows per glyph, 2 ASCII hexadecimal digits per byte,
101 so divide number of digits by 32 (shift right 5 bits).
102 */
103       glyph_width[loc] = (strlen (gstart) - 1) » 5;
104     }
105     else if ((loc >= PIKTO_START) && (loc <= PIKTO_END)) {
106       gstart = strchr (teststring,':') + 1;
107       pikto_width[loc - PIKTO_START] = strlen (gstart) <= 34 ? 1 : 2;
108     }
109   }
110
111   fclose (infilefp);
112
113   /*
114 Now read the combining character code points.  These have width of 0.
115 */
116   if ((infilefp = fopen (argv[2],"r")) == NULL) {
117     fprintf (stderr,"ERROR - combining characters file %s not found.\n\n", argv[2]);
118     exit (EXIT_FAILURE);
119   }
120
121   while (fgets (teststring, MAXSTRING-1, infilefp) != NULL) {
122     sscanf (teststring, "%X:%*s", &loc);
123     if (loc < 0x20000) glyph_width[loc] = 0;
124   }
125
126   fclose (infilefp);
```

```
127
128   /*
129 Code Points with Unusual Properties (Unicode Standard, Chapter 4).
130
131 As of Unifont 10.0.04, use the widths in the "*-nonprinting.hex"
132 files.  If an application is smart enough to know how to handle
133 these special cases, it will not render the "nonprinting" glyph
134 and will treat the code point as being zero-width.
135 */
136 // glyph_width[0]=0; /* NULL character */
137 // for (i = 0x0001; i <= 0x001F; i++) glyph_width[i]=-1; /* Control Characters */
138 // for (i = 0x007F; i <= 0x009F; i++) glyph_width[i]=-1; /* Control Characters */
139
140 // glyph_width[0x034F]=0; /* combining grapheme joiner             */
141 // glyph_width[0x180B]=0; /* Mongolian free variation selector one  */
142 // glyph_width[0x180C]=0; /* Mongolian free variation selector two  */
143 // glyph_width[0x180D]=0; /* Mongolian free variation selector three */
144 // glyph_width[0x180E]=0; /* Mongolian vowel separator             */
145 // glyph_width[0x200B]=0; /* zero width space                      */
146 // glyph_width[0x200C]=0; /* zero width non-joiner                 */
147 // glyph_width[0x200D]=0; /* zero width joiner                     */
148 // glyph_width[0x200E]=0; /* left-to-right mark                    */
149 // glyph_width[0x200F]=0; /* right-to-left mark                    */
150 // glyph_width[0x202A]=0; /* left-to-right embedding               */
151 // glyph_width[0x202B]=0; /* right-to-left embedding               */
152 // glyph_width[0x202C]=0; /* pop directional formatting            */
153 // glyph_width[0x202D]=0; /* left-to-right override                */
154 // glyph_width[0x202E]=0; /* right-to-left override                */
155 // glyph_width[0x2060]=0; /* word joiner                           */
156 // glyph_width[0x2061]=0; /* function application                  */
157 // glyph_width[0x2062]=0; /* invisible times                       */
158 // glyph_width[0x2063]=0; /* invisible separator                   */
159 // glyph_width[0x2064]=0; /* invisible plus                        */
160 // glyph_width[0x206A]=0; /* inhibit symmetric swapping            */
161 // glyph_width[0x206B]=0; /* activate symmetric swapping           */
162 // glyph_width[0x206C]=0; /* inhibit arabic form shaping           */
163 // glyph_width[0x206D]=0; /* activate arabic form shaping          */
164 // glyph_width[0x206E]=0; /* national digit shapes                 */
165 // glyph_width[0x206F]=0; /* nominal digit shapes                  */
166
167 // /* Variation Selector-1 to Variation Selector-16 */
168 // for (i = 0xFE00; i <= 0xFE0F; i++) glyph_width[i] = 0;
169
170 // glyph_width[0xFEFF]=0; /* zero width no-break space         */
171 // glyph_width[0xFFF9]=0; /* interlinear annotation anchor     */
172 // glyph_width[0xFFFA]=0; /* interlinear annotation separator  */
173 // glyph_width[0xFFFB]=0; /* interlinear annotation terminator */
174   /*
175 Let glyph widths represent 0xFFFC (object replacement character)
176 and 0xFFFD (replacement character).
177 */
178
179   /*
180 Hangul Jamo:
181
182 Leading Consonant (Choseong): leave spacing as is.
183
184 Hangul Choseong Filler (U+115F): set width to 2.
185
186 Hangul Jungseong Filler, Hangul Vowel (Jungseong), and
187 Final Consonant (Jongseong): set width to 0, because these
188 combine with the leading consonant as one composite syllabic
189 glyph.  As of Unicode 5.2, the Hangul Jamo block (U+1100..U+11FF)
190 is completely filled.
191 */
192   // for (i = 0x1160; i <= 0x11FF; i++) glyph_width[i]=0; /* Vowels & Final Consonants */
193
194   /*
195 Private Use Area -- the width is undefined, but likely
196 to be 2 charcells wide either from a graphic glyph or
197 from a four-digit hexadecimal glyph representing the
198 code point.  Therefore if any PUA glyph does not have
199 a non-zero width yet, assign it a default width of 2.
200 The Unicode Standard allows giving PUA characters
201 default property values; see for example The Unicode
202 Standard Version 5.0, p. 91.  This same default is
203 used for higher plane PUA code points below.
204 */
205   // for (i = 0xE000; i <= 0xF8FF; i++) {
206   //    if (glyph_width[i] == 0) glyph_width[i]=2;
207   // }
```

```
208
209   /*
210 <not a character>
211 */
212   for (i = 0xFDD0; i <= 0xFDEF; i++) glyph_width[i] = -1;
213   glyph_width[0xFFFE] = -1; /* Byte Order Mark */
214   glyph_width[0xFFFF] = -1; /* Byte Order Mark */
215
216   /* Surrogate Code Points */
217   for (i = 0xD800; i <= 0xDFFF; i++) glyph_width[i]=-1;
218
219   /* CJK Code Points */
220   for (i = 0x4E00; i <= 0x9FFF; i++) if (glyph_width[i] < 0) glyph_width[i] = 2;
221   for (i = 0x3400; i <= 0x4DBF; i++) if (glyph_width[i] < 0) glyph_width[i] = 2;
222   for (i = 0xF900; i <= 0xFAFF; i++) if (glyph_width[i] < 0) glyph_width[i] = 2;
223
224   /*
225 Now generate the output file.
226 */
227   printf ("/*\n");
228   printf ("   wcwidth and wcswidth functions, as per IEEE 1003.1-2008\n");
229   printf ("   System Interfaces, pp. 2241 and 2251.\n\n");
230   printf ("   Author: Paul Hardy, 2013\n\n");
231   printf ("   Copyright (c) 2013 Paul Hardy\n\n");
232   printf ("   LICENSE:\n");
233   printf ("\n");
234   printf ("      This program is free software: you can redistribute it and/or modify\n");
235   printf ("      it under the terms of the GNU General Public License as published by\n");
236   printf ("      the Free Software Foundation, either version 2 of the License, or\n");
237   printf ("      (at your option) any later version.\n");
238   printf ("\n");
239   printf ("      This program is distributed in the hope that it will be useful,\n");
240   printf ("      but WITHOUT ANY WARRANTY; without even the implied warranty of\n");
241   printf ("      MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the\n");
242   printf ("      GNU General Public License for more details.\n");
243   printf ("\n");
244   printf ("      You should have received a copy of the GNU General Public License\n");
245   printf ("      along with this program.  If not, see <http://www.gnu.org/licenses/>.\n");
246   printf ("*/\n\n");
247
248   printf ("#include <wchar.h>\n\n");
249   printf ("/* Definitions for Pikto CSUR Private Use Area glyphs */\n");
250   printf ("#define PIKTO_START\t0x%06X\n", PIKTO_START);
251   printf ("#define PIKTO_END\t0x%06X\n", PIKTO_END);
252   printf ("#define PIKTO_SIZE\t(PIKTO_END - PIKTO_START + 1)\n");
253   printf ("\n\n");
254   printf ("/* wcwidth -- return charcell positions of one code point */\n");
255   printf ("inline int\nwcwidth (wchar_t wc)\n{\n");
256   printf ("   return (wcswidth (&wc, 1));\n");
257   printf ("}\n");
258   printf ("\n\n");
259   printf ("int\nwcswidth (const wchar_t *pwcs, size_t n)\n{\n\n");
260   printf ("   int i;                  /* loop variable                                  */\n");
261   printf ("   unsigned codept;        /* Unicode code point of current character        */\n");
262   printf ("   unsigned plane;         /* Unicode plane, 0x00..0x10                      */\n");
263   printf ("   unsigned lower17;       /* lower 17 bits of Unicode code point            */\n");
264   printf ("   unsigned lower16;       /* lower 16 bits of Unicode code point            */\n");
265   printf ("   int lowpt, midpt, highpt; /* for binary searching in plane1zeroes[]       */\n");
266   printf ("   int found;              /* for binary searching in plane1zeroes[]         */\n");
267   printf ("   int totalwidth;         /* total width of string, in charcells (1 or 2/glyph) */\n");
268   printf ("   int illegalchar;        /* Whether or not this code point is illegal      */\n");
269   putchar ('\n');
270
271   /*
272 Print the glyph_width[] array for glyphs widths in the
273 Basic Multilingual Plane (Plane 0).
274 */
275   printf ("   char glyph_width[0x20000] = {");
276   for (i = 0; i < 0x10000; i++) {
277     if ((i & 0x1F) == 0)
278       printf ("\n      /* U+%04X */ ", i);
279     printf ("%d,", glyph_width[i]);
280   }
281   for (i = 0x10000; i < 0x20000; i++) {
282     if ((i & 0x1F) == 0)
283       printf ("\n      /* U+%06X */ ", i);
284     printf ("%d", glyph_width[i]);
285     if (i < 0x1FFFF) putchar (',');
286   }
287   printf ("\n   };\n\n");
288
```

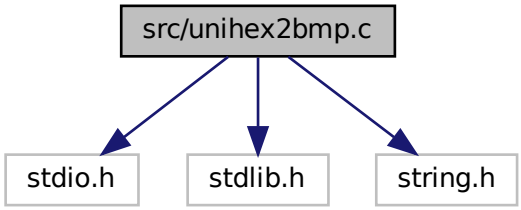
```
289   /*
290 Print the pikto_width[] array for Pikto glyph widths.
291 */
292   printf ("   char pikto_width[PIKTO_SIZE] = {");
293   for (i = 0; i < PIKTO_SIZE; i++) {
294     if ((i & 0x1F) == 0)
295       printf ("\n      /* U+%06X */ ", PIKTO_START + i);
296     printf ("%d", pikto_width[i]);
297     if ((PIKTO_START + i) < PIKTO_END) putchar (',');
298   }
299   printf ("\n   };\n\n");
300
301   /*
302 Execution part of wcswidth.
303 */
304   printf ("\n");
305   printf ("   illegalchar = totalwidth = 0;\n");
306   printf ("   for (i = 0; !illegalchar && i < n; i++) {\n");
307   printf ("     codept  = pwcs[i];\n");
308   printf ("     plane   = codept » 16;\n");
309   printf ("     lower17 = codept & 0x1FFFF;\n");
310   printf ("     lower16 = codept & 0xFFFF;\n");
311   printf ("     if (plane < 2) { /* the most common case */\n");
312   printf ("       if (glyph_width[lower17] < 0) illegalchar = 1;\n");
313   printf ("       else totalwidth += glyph_width[lower17];\n");
314   printf ("     }\n");
315   printf ("     else { /* a higher plane or beyond Unicode range */\n");
316   printf ("       if  ((lower16 == 0xFFFE) || (lower16 == 0xFFFF)) {\n");
317   printf ("         illegalchar = 1;\n");
318   printf ("       }\n");
319   printf ("       else if (plane < 4) {  /* Ideographic Plane */\n");
320   printf ("         totalwidth += 2; /* Default ideographic width */\n");
321   printf ("       }\n");
322   printf ("       else if (plane == 0x0F) {  /* CSUR Private Use Area */\n");
323   printf ("         if (lower16 <= 0x0E6F) { /* Kinya */\n");
324   printf ("           totalwidth++; /* all Kinya syllables have width 1 */\n");
325   printf ("         }\n");
326   printf ("         else if (lower16 <= (PIKTO_END & 0xFFFF)) { /* Pikto */\n");
327   printf ("           if (pikto_width[lower16 - (PIKTO_START & 0xFFFF)] < 0) illegalchar = 1;\n");
328   printf ("           else totalwidth += pikto_width[lower16 - (PIKTO_START & 0xFFFF)];\n");
329   printf ("         }\n");
330   printf ("       }\n");
331   printf ("       else if (plane > 0x10) {\n");
332   printf ("         illegalchar = 1;\n");
333   printf ("       }\n");
334   printf ("       /* Other non-printing in higher planes; return -1 as per IEEE 1003.1-2008. */\n");
335   printf ("       else if (/* language tags */\n");
336   printf ("              codept == 0x0E0001 || (codept >= 0x0E0020 && codept <= 0x0E007F) ||\n");
337   printf ("              /* variation selectors, 0x0E0100..0x0E01EF */\n");
338   printf ("              (codept >= 0x0E0100 && codept <= 0x0E01EF)) {\n");
339   printf ("         illegalchar = 1;\n");
340   printf ("       }\n");
341   printf ("       /*\n");
342   printf ("         Unicode plane 0x02..0x10 printing character\n");
343   printf ("       */\n");
344   printf ("       else {\n");
345   printf ("         illegalchar = 1; /* code is not in font */\n");
346   printf ("       }\n");
347   printf ("\n");
348   printf ("     }\n");
349   printf ("   }\n");
350   printf ("   if (illegalchar) totalwidth = -1;\n");
351   printf ("\n");
352   printf ("   return (totalwidth);\n");
353   printf ("\n");
354   printf ("}\n");
355
356   exit (EXIT_SUCCESS);
357 }
```

## 5.13 src/unihex2bmp.c File Reference

unihex2bmp - Turn a GNU Unifont hex glyph page of 256 code points into a bitmap for editing

#include <stdio.h>
#include <stdlib.h>

#include <string.h>
Include dependency graph for unihex2bmp.c:




## Macros

- #define MAXBUF 256

## Functions

- int main (int argc, char *argv[])

  The main function.
- int hex2bit (char *instring, unsigned char character[32][4])

  Generate a bitmap for one glyph.
- int init (unsigned char bitmap[17 *32][18 *4])

  Initialize the bitmap grid.

## Variables

- char * hex [18]

  GNU Unifont bitmaps for hexadecimal digits.
- unsigned char hexbits [18][32]

  The digits converted into bitmaps.
- unsigned unipage =0

  Unicode page number, 0x00..0xff.
- int flip =1

  Transpose entire matrix as in Unicode book.

### 5.13.1 Detailed Description

unihex2bmp - Turn a GNU Unifont hex glyph page of 256 code points into a bitmap for editing

Author

Paul Hardy, unifoundry <at> unifoundry.com, December 2007

Copyright



This program reads in a GNU Unifont .hex file, extracts a range of 256 code points, and converts it a Microsoft Bitmap (.bmp) or Wireless Bitmap file.

Synopsis: unihex2bmp [-iin_file.hex] [-oout_file.bmp] [-f] [-phex_page_num] [-w]

### 5.13.2 Function Documentation

#### 5.13.2.1 hex2bit()

```
int hex2bit (
            char * instring,
            unsigned char character[32][4] )
```

Generate a bitmap for one glyph.

Convert the portion of a hex string after the ':' into a character bitmap.

If string is >= 128 characters, it will fill all 4 bytes per row. If string is >= 64 characters and < 128, it will fill 2 bytes per row. Otherwise, it will fill 1 byte per row.

Parameters

| | | |
|---|---|---|
| in | instring | The character array containing the glyph bitmap. |
| out | character | Glyph bitmap, 8, 16, or 32 columns by 16 rows tall. |

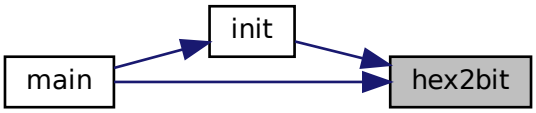
Returns

Always returns 0.

Definition at line 361 of file unihex2bmp.c.
362 {

```
363
364   int i;  /* current row in bitmap character */
365   int j;  /* current character in input string */
366   int k;  /* current byte in bitmap character */
367   int width;  /* number of output bytes to fill - 1: 0, 1, 2, or 3 */
368
369   for (i=0; i<32; i++)  /* erase previous character */
370     character[i][0] = character[i][1] = character[i][2] = character[i][3] = 0;
371   j=0;  /* current location is at beginning of instring */
372
373   if (strlen (instring) <= 34)  /* 32 + possible '\r', '\n' */
374     width = 0;
375   else if (strlen (instring) <= 66)  /* 64 + possible '\r', '\n' */
376     width = 1;
377   else if (strlen (instring) <= 98)  /* 96 + possible '\r', '\n' */
378     width = 3;
379   else  /* the maximum allowed is quadruple-width */
380     width = 4;
381
382   k = (width > 1) ? 0 : 1; /* if width > double, start at index 1 else at 0 */
383
384   for (i=8; i<24; i++) {  /* 16 rows per input character, rows 8..23 */
385     sscanf (&instring[j], "%2hhx", &character[i][k]);
386     j += 2;
387     if (width > 0) { /* add next pair of hex digits to this row */
388       sscanf (&instring[j], "%2hhx", &character[i][k+1]);
389       j += 2;
390       if (width > 1) { /* add next pair of hex digits to this row */
391         sscanf (&instring[j], "%2hhx", &character[i][k+2]);
392         j += 2;
393         if (width > 2) {  /* quadruple-width is maximum width */
394           sscanf (&instring[j], "%2hhx", &character[i][k+3]);
395           j += 2;
396         }
397       }
398     }
399   }
400
401   return (0);
402 }
```

Here is the caller graph for this function:




#### 5.13.2.2 init()

```
int init (
                unsigned char bitmap[17 *32][18 *4] )
```

Initialize the bitmap grid.

Parameters

| | | |
|---|---|---|
| out | bitmap | The bitmap to generate, with 32x32 pixel glyph areas. |

Returns

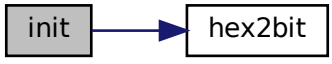
Always returns 0.

Definition at line 412 of file unihex2bmp.c.

```
413 {
414   int i, j;
415   unsigned char charbits[32][4];  /* bitmap for one character, 4 bytes/row */
416   unsigned toppixelrow;
417   unsigned thiscol;
418   unsigned char pnybble0, pnybble1, pnybble2, pnybble3;
419
420   for (i=0; i<18; i++) { /* bitmaps for '0'..'9', 'A'-'F', 'u', '+' */
421
422     hex2bit (&hex[i][5], charbits);  /* convert hex string to 32*4 bitmap */
423
424     for (j=0; j<32; j++) hexbits[i][j] = ~charbits[j][1];
425   }
426
427   /*
428 Initialize bitmap to all white.
429 */
430   for (toppixelrow=0; toppixelrow < 17*32; toppixelrow++) {
431     for (thiscol=0; thiscol<18; thiscol++) {
432       bitmap[toppixelrow][(thiscol « 2)    ] = 0xff;
433       bitmap[toppixelrow][(thiscol « 2) | 1] = 0xff;
434       bitmap[toppixelrow][(thiscol « 2) | 2] = 0xff;
435       bitmap[toppixelrow][(thiscol « 2) | 3] = 0xff;
436     }
437   }
438   /*
439 Write the "u+nnnn" table header in the upper left-hand corner,
440 where nnnn is the upper 16 bits of a 32-bit Unicode assignment.
441 */
442   pnybble3 = (unipage » 20);
443   pnybble2 = (unipage » 16) & 0xf;
444   pnybble1 = (unipage » 12) & 0xf;
445   pnybble0 = (unipage »  8) & 0xf;
446   for (i=0; i<32; i++) {
447     bitmap[i][1] = hexbits[16][i];  /* copy 'u' */
448     bitmap[i][2] = hexbits[17][i];  /* copy '+' */
449     bitmap[i][3] = hexbits[pnybble3][i];
450     bitmap[i][4] = hexbits[pnybble2][i];
451     bitmap[i][5] = hexbits[pnybble1][i];
452     bitmap[i][6] = hexbits[pnybble0][i];
453   }
454   /*
455 Write low-order 2 bytes of Unicode number assignments, as hex labels
456 */
457   pnybble3 = (unipage » 4) & 0xf;  /* Highest-order hex digit */
458   pnybble2 = (unipage    ) & 0xf;  /* Next highest-order hex digit */
459   /*
460 Write the column headers in bitmap[][] (row headers if flipped)
461 */
462   toppixelrow = 32 * 17 - 1; /* maximum pixel row number */
463   /*
464 Label the column headers. The hexbits[][] bytes are split across two
465 bitmap[][] entries to center a the hex digits in a column of 4 bytes.
466 OR highest byte with 0xf0 and lowest byte with 0x0f to make outer
467 nybbles white (0=black, 1-white).
468 */
469   for (i=0; i<16; i++) {
470     for (j=0; j<32; j++) {
471       if (flip) {  /* transpose matrix */
472         bitmap[j][((i+2) « 2) | 0]  = (hexbits[pnybble3][j] » 4) | 0xf0;
473         bitmap[j][((i+2) « 2) | 1]  = (hexbits[pnybble3][j] « 4) |
474                                       (hexbits[pnybble2][j] » 4);
475         bitmap[j][((i+2) « 2) | 2]  = (hexbits[pnybble2][j] « 4) |
476                                       (hexbits[i][j] » 4);
477         bitmap[j][((i+2) « 2) | 3]  = (hexbits[i][j] « 4) | 0x0f;
```

```
478         }
479         else {
480            bitmap[j][((i+2) « 2) | 1] = (hexbits[i][j] » 4) | 0xf0;
481            bitmap[j][((i+2) « 2) | 2] = (hexbits[i][j] « 4) | 0x0f;
482         }
483      }
484   }
485   /*
486 Now use the single hex digit column graphics to label the row headers.
487 */
488   for (i=0; i<16; i++) {
489      toppixelrow = 32 * (i + 1) - 1; /* from bottom to top    */
490
491      for (j=0; j<32; j++) {
492         if (!flip) {  /* if not transposing matrix */
493            bitmap[toppixelrow + j][4] = hexbits[pnybble3][j];
494            bitmap[toppixelrow + j][5] = hexbits[pnybble2][j];
495         }
496         bitmap[toppixelrow + j][6] = hexbits[i][j];
497      }
498   }
499   /*
500 Now draw grid lines in bitmap, around characters we just copied.
501 */
502   /* draw vertical lines 2 pixels wide */
503   for (i=1*32; i<17*32; i++) {
504      if ((i & 0x1f) == 7)
505         i++;
506      else if ((i & 0x1f) == 14)
507         i += 2;
508      else if ((i & 0x1f) == 22)
509         i++;
510      for (j=1; j<18; j++) {
511         bitmap[i][(j « 2) | 3] &= 0xfe;
512      }
513   }
514   /* draw horizontal lines 1 pixel tall */
515   for (i=1*32-1; i<18*32-1; i+=32) {
516      for (j=2; j<18; j++) {
517         bitmap[i][(j « 2)    ] = 0x00;
518         bitmap[i][(j « 2) | 1] = 0x81;
519         bitmap[i][(j « 2) | 2] = 0x81;
520         bitmap[i][(j « 2) | 3] = 0x00;
521      }
522   }
523   /* fill in top left corner pixel of grid */
524   bitmap[31][7] = 0xfe;
525
526   return (0);
527 }
```

Here is the call graph for this function:



Here is the caller graph for this function:


#### 5.13.2.3 main()

```
int main (
                int argc,
                char * argv[] )
```

The main function.

Parameters

| | | |
|---|---|---|
| in | argc | The count of command line arguments. |
| in | argv | Pointer to array of command line arguments. |

Returns

This program exits with status 0.

Definition at line 96 of file unihex2bmp.c.

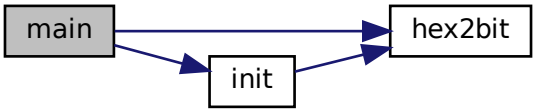
```
97 {
98
99   int i, j;                /* loop variables                    */
100   unsigned k0;             /* temp Unicode char variable        */
101   unsigned swap;           /* temp variable for swapping values */
102   char inbuf[256];         /* input buffer                      */
103   unsigned filesize;       /* size of file in bytes             */
104   unsigned bitmapsize;     /* size of bitmap image in bytes     */
105   unsigned thischar;       /* the current character             */
106   unsigned char thischarbyte; /* unsigned char lowest byte of Unicode char */
107   int thischarrow;         /* row 0..15 where this character belongs  */
108   int thiscol;             /* column 0..15 where this character belongs */
109   int toppixelrow;         /* pixel row, 0..16*32-1             */
110   unsigned lastpage=0;     /* the last Unicode page read in font file */
111   int wbmp=0;              /* set to 1 if writing .wbmp format file */
112
113   unsigned char bitmap[17*32][18*4]; /* final bitmap */
114   unsigned char charbits[32][4];  /* bitmap for one character, 4 bytes/row */
115
116   char *infile="", *outfile="";  /* names of input and output files */
117   FILE *infp, *outfp;      /* file pointers of input and output files */
118
119   int init();              /* initializes bitmap row/col labeling, &c. */
120   int hex2bit();           /* convert hex string --> bitmap */
121
```

```
122   bitmapsize = 17*32*18*4;  /* 17 rows by 18 cols, each 4 bytes */
123
124   if (argc > 1) {
125      for (i = 1; i < argc; i++) {
126         if (argv[i][0] == '-') {  /* this is an option argument */
127            switch (argv[i][1]) {
128               case 'f':  /* flip (transpose) glyphs in bitmap as in standard */
129                  flip = !flip;
130                  break;
131               case 'i':  /* name of input file */
132                  infile = &argv[i][2];
133                  break;
134               case 'o':  /* name of output file */
135                  outfile = &argv[i][2];
136                  break;
137               case 'p':  /* specify a Unicode page other than default of 0 */
138                  sscanf (&argv[i][2], "%x", &unipage); /* Get Unicode page */
139                  break;
140               case 'w':  /* write a .wbmp file instead of a .bmp file */
141                  wbmp = 1;
142                  break;
143               default:    /* if unrecognized option, print list and exit */
144                  fprintf (stderr, "\nSyntax:\n\n");
145                  fprintf (stderr, "    %s -p<Unicode_Page> ", argv[0]);
146                  fprintf (stderr, "-i<Input_File> -o<Output_File> -w\n\n");
147                  fprintf (stderr, "    -w specifies .wbmp output instead of ");
148                  fprintf (stderr, "default Windows .bmp output.\n\n");
149                  fprintf (stderr, "    -p is followed by 1 to 6 ");
150                  fprintf (stderr, "Unicode page hex digits ");
151                  fprintf (stderr, "(default is Page 0).\n\n");
152                  fprintf (stderr, "\nExample:\n\n");
153                  fprintf (stderr, "    %s -p83 -iunifont.hex -ou83.bmp\n\n\n",
154                        argv[0]);
155                  exit (1);
156            }
157         }
158      }
159   }
160   /*
161 Make sure we can open any I/O files that were specified before
162 doing anything else.
163 */
164   if (strlen (infile) > 0) {
165      if ((infp = fopen (infile, "r")) == NULL) {
166         fprintf (stderr, "Error: can't open %s for input.\n", infile);
167         exit (1);
168      }
169   }
170   else {
171      infp = stdin;
172   }
173   if (strlen (outfile) > 0) {
174      if ((outfp = fopen (outfile, "w")) == NULL) {
175         fprintf (stderr, "Error: can't open %s for output.\n", outfile);
176         exit (1);
177      }
178   }
179   else {
180      outfp = stdout;
181   }
182
183   (void)init(bitmap); /* initialize bitmap with row/column headers, etc. */
184
185   /*
186 Read in the characters in the page
187 */
188   while (lastpage <= unipage && fgets (inbuf, MAXBUF-1, infp) != NULL) {
189      sscanf (inbuf, "%x", &thischar);
190      lastpage = thischar » 8; /* keep Unicode page to see if we can stop */
191      if (lastpage == unipage) {
192         thischarbyte = (unsigned char)(thischar & 0xff);
193         for (k0=0; inbuf[k0] != ':'; k0++);
194         k0++;
195         hex2bit (&inbuf[k0], charbits);  /* convert hex string to 32*4 bitmap */
196
197         /*
198 Now write character bitmap upside-down in page array, to match
199 .bmp file order.  In the .wbmp‘ and .bmp files, white is a '1'
200 bit and black is a '0' bit, so complement charbits[][].
201 */
202
```

```
203         thiscol = (thischarbyte & 0xf) + 2;  /* column number will be 1..16  */
204         thischarrow = thischarbyte » 4;     /* charcter row number, 0..15   */
205         if (flip) {  /* swap row and column placement */
206           swap = thiscol;
207           thiscol = thischarrow;
208           thischarrow = swap;
209           thiscol += 2;       /* column index starts at 1 */
210           thischarrow -= 2;   /* row index starts at 0    */
211         }
212         toppixelrow = 32 * (thischarrow + 1) - 1; /* from bottom to top    */
213
214         /*
215 Copy the center of charbits[][] because hex characters only
216 occupy rows 8 to 23 and column byte 2 (and for 16 bit wide
217 characters, byte 3).  The charbits[][] array was given 32 rows
218 and 4 column bytes for completeness in the beginning.
219 */
220         for (i=8; i<24; i++) {
221           bitmap[toppixelrow + i][(thiscol « 2) | 0] =
222             ~charbits[i][0] & 0xff;
223           bitmap[toppixelrow + i][(thiscol « 2) | 1] =
224             ~charbits[i][1] & 0xff;
225           bitmap[toppixelrow + i][(thiscol « 2) | 2] =
226             ~charbits[i][2] & 0xff;
227           /* Only use first 31 bits; leave vertical rule in 32nd column */
228           bitmap[toppixelrow + i][(thiscol « 2) | 3] =
229             ~charbits[i][3] & 0xfe;
230         }
231         /*
232 Leave white space in 32nd column of rows 8, 14, 15, and 23
233 to leave 16 pixel height upper, middle, and lower guides.
234 */
235         bitmap[toppixelrow +  8][(thiscol « 2) | 3] |= 1;
236         bitmap[toppixelrow + 14][(thiscol « 2) | 3] |= 1;
237         bitmap[toppixelrow + 15][(thiscol « 2) | 3] |= 1;
238         bitmap[toppixelrow + 23][(thiscol « 2) | 3] |= 1;
239       }
240     }
241     /*
242 Now write the appropriate bitmap file format, either
243 Wireless Bitmap or Microsoft Windows bitmap.
244 */
245     if (wbmp) {  /* Write a Wireless Bitmap .wbmp format file */
246       /*
247 Write WBMP header
248 */
249       fprintf (outfp, "%c", 0x00); /* Type of image; always 0 (monochrome) */
250       fprintf (outfp, "%c", 0x00); /* Reserved; always 0                   */
251       fprintf (outfp, "%c%c", 0x84, 0x40); /* Width  = 576 pixels          */
252       fprintf (outfp, "%c%c", 0x84, 0x20); /* Height = 544 pixels          */
253       /*
254 Write bitmap image
255 */
256       for (toppixelrow=0; toppixelrow <= 17*32-1; toppixelrow++) {
257         for (j=0; j<18; j++) {
258           fprintf (outfp, "%c", bitmap[toppixelrow][(j«2)    ]);
259           fprintf (outfp, "%c", bitmap[toppixelrow][(j«2) | 1]);
260           fprintf (outfp, "%c", bitmap[toppixelrow][(j«2) | 2]);
261           fprintf (outfp, "%c", bitmap[toppixelrow][(j«2) | 3]);
262         }
263       }
264     }
265     else {  /* otherwise, write a Microsoft Windows .bmp format file */
266       /*
267 Write the .bmp file -- start with the header, then write the bitmap
268 */
269
270       /* 'B', 'M' appears at start of every .bmp file */
271       fprintf (outfp, "%c%c", 0x42, 0x4d);
272
273       /* Write file size in bytes */
274       filesize   = 0x3E + bitmapsize;
275       fprintf (outfp, "%c", (unsigned char)((filesize        ) & 0xff));
276       fprintf (outfp, "%c", (unsigned char)((filesize » 0x08) & 0xff));
277       fprintf (outfp, "%c", (unsigned char)((filesize » 0x10) & 0xff));
278       fprintf (outfp, "%c", (unsigned char)((filesize » 0x18) & 0xff));
279
280       /* Reserved - 0's */
281       fprintf (outfp, "%c%c%c%c", 0x00, 0x00, 0x00, 0x00);
282
283       /* Offset from start of file to bitmap data */
```

```
284       fprintf (outfp, "%c%c%c%c", 0x3E, 0x00, 0x00, 0x00);
285
286       /* Length of bitmap info header */
287       fprintf (outfp, "%c%c%c%c", 0x28, 0x00, 0x00, 0x00);
288
289       /* Width of bitmap in pixels */
290       fprintf (outfp, "%c%c%c%c", 0x40, 0x02, 0x00, 0x00);
291
292       /* Height of bitmap in pixels */
293       fprintf (outfp, "%c%c%c%c", 0x20, 0x02, 0x00, 0x00);
294
295       /* Planes in bitmap (fixed at 1) */
296       fprintf (outfp, "%c%c", 0x01, 0x00);
297
298       /* bits per pixel (1 = monochrome) */
299       fprintf (outfp, "%c%c", 0x01, 0x00);
300
301       /* Compression (0 = none) */
302       fprintf (outfp, "%c%c%c%c", 0x00, 0x00, 0x00, 0x00);
303
304       /* Size of bitmap data in bytes */
305       fprintf (outfp, "%c", (unsigned char)((bitmapsize        ) & 0xff));
306       fprintf (outfp, "%c", (unsigned char)((bitmapsize » 0x08) & 0xff));
307       fprintf (outfp, "%c", (unsigned char)((bitmapsize » 0x10) & 0xff));
308       fprintf (outfp, "%c", (unsigned char)((bitmapsize » 0x18) & 0xff));
309
310       /* Horizontal resolution in pixels per meter */
311       fprintf (outfp, "%c%c%c%c", 0xC4, 0x0E, 0x00, 0x00);
312
313       /* Vertical resolution in pixels per meter */
314       fprintf (outfp, "%c%c%c%c", 0xC4, 0x0E, 0x00, 0x00);
315
316       /* Number of colors used */
317       fprintf (outfp, "%c%c%c%c", 0x02, 0x00, 0x00, 0x00);
318
319       /* Number of important colors */
320       fprintf (outfp, "%c%c%c%c", 0x02, 0x00, 0x00, 0x00);
321
322       /* The color black: B=0x00, G=0x00, R=0x00, Filler=0xFF */
323       fprintf (outfp, "%c%c%c%c", 0x00, 0x00, 0x00, 0x00);
324
325       /* The color white: B=0xFF, G=0xFF, R=0xFF, Filler=0xFF */
326       fprintf (outfp, "%c%c%c%c", 0xFF, 0xFF, 0xFF, 0x00);
327
328       /*
329  Now write the raw data bits.  Data is written from the lower
330  left-hand corner of the image to the upper right-hand corner
331  of the image.
332  */
333       for (toppixelrow=17*32-1; toppixelrow >= 0; toppixelrow--) {
334          for (j=0; j<18; j++) {
335             fprintf (outfp, "%c", bitmap[toppixelrow][(j«2)    ]);
336             fprintf (outfp, "%c", bitmap[toppixelrow][(j«2) | 1]);
337             fprintf (outfp, "%c", bitmap[toppixelrow][(j«2) | 2]);
338
339             fprintf (outfp, "%c", bitmap[toppixelrow][(j«2) | 3]);
340          }
341       }
342    }
343    exit (0);
344 }
```

Here is the call graph for this function:



### 5.13.3 Variable Documentation

#### 5.13.3.1 hex

char* hex[18]

Initial value:
```
= {
    "0030:00000000182442424242424224180000",
    "0031:000000000818280808080808083E0000",
    "0032:000000003C4242020C102040407E0000",
    "0033:000000003C4242021C020242423C0000",
    "0034:00000000040C142444447E0404040000",
    "0035:000000007E4040407C020202423C0000",
    "0036:000000001C2040407C424242423C0000",
    "0037:000000007E0202040404080808080000",
    "0038:000000003C4242423C424242423C0000",
    "0039:000000003C4242423E02020204380000",
    "0041:0000000018242442427E424242420000",
    "0042:000000007C4242427C424242427C0000",
    "0043:000000003C42424040404042423C0000",
    "0044:00000000784442424242424244780000",
    "0045:000000007E4040407C404040407E0000",
    "0046:000000007E4040407C40404040400000",
    "0055:00000000424242424242424242423C0000",
    "002B:000000000000000808087F080808000000"
  }
```

GNU Unifont bitmaps for hexadecimal digits.

These are the GNU Unifont hex strings for '0'-'9' and 'A'-'F', for encoding as bit strings in row and column headers.

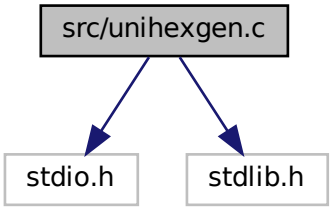
Looking at the final bitmap as a grid of 32*32 bit tiles, the first row contains a hexadecimal character string of the first 3 hex digits in a 4 digit Unicode character name; the top column contains a hex character string of the 4th (low-order) hex digit of the Unicode character.

Definition at line 62 of file unihex2bmp.c.

## 5.14 src/unihexgen.c File Reference

unihexgen - Generate a series of glyphs containing hexadecimal code points

#include <stdio.h>
#include <stdlib.h>
Include dependency graph for unihexgen.c:



## Functions

- int main (int argc, char *argv[])
  The main function.
- void hexprint4 (int thiscp)
  Generate a bitmap containing a 4-digit Unicode code point.
- void hexprint6 (int thiscp)
  Generate a bitmap containing a 6-digit Unicode code point.

## Variables

- char hexdigit [16][5]
  Bitmap pattern for each hexadecimal digit.

### 5.14.1 Detailed Description

unihexgen - Generate a series of glyphs containing hexadecimal code points

Author

Paul Hardy

Copyright

Copyright (C) 2013 Paul Hardy

This program generates glyphs in Unifont .hex format that contain four- or six-digit hexadecimal numbers in a 16x16 pixel area. These are rendered as white digits on a black background.

argv[1] is the starting code point (as a hexadecimal string, with no leading "0x".

argv[2] is the ending code point (as a hexadecimal string, with no leading "0x".

For example:

```
unihexgen e000 f8ff > pua.hex
```

This generates the Private Use Area glyph file.

This utility program works in Roman Czyborra's unifont.hex file format, the basis of the GNU Unifont package.

### 5.14.2 Function Documentation

#### 5.14.2.1 hexprint4()

```
void hexprint4 (
            int thiscp )
```

Generate a bitmap containing a 4-digit Unicode code point.

Takes a 4-digit Unicode code point as an argument and prints a unifont.hex string for it to stdout.

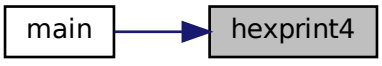
Parameters

| | | |
|---|---|---|
| in | thiscp | The current code point for which to generate a glyph. |

Definition at line 160 of file unihexgen.c.

```
161 {
162
163   int grid[16]; /* the glyph grid we'll build */
164
165   int row;      /* row number in current glyph */
166   int digitrow; /* row number in current hex digit being rendered */
167   int rowbits;  /* 1 & 0 bits to draw current glyph row */
168
169   int d1, d2, d3, d4; /* four hexadecimal digits of each code point */
170
171   d1 = (thiscp » 12) & 0xF;
172   d2 = (thiscp »  8) & 0xF;
173   d3 = (thiscp »  4) & 0xF;
174   d4 = (thiscp      ) & 0xF;
175
176   /* top and bottom rows are white */
177   grid[0] = grid[15] = 0x0000;
178
179   /* 14 inner rows are 14-pixel wide black lines, centered */
180   for (row = 1; row < 15; row++) grid[row] = 0x7FFE;
181
182   printf ("%04X:", thiscp);
183
184   /*
185 Render the first row of 2 hexadecimal digits
186 */
187   digitrow = 0; /* start at top of first row of digits to render */
188   for (row = 2; row < 7; row++) {
189     rowbits = (hexdigit[d1][digitrow] « 9) |
190               (hexdigit[d2][digitrow] « 3);
191     grid[row] ^= rowbits; /* digits appear as white on black background */
192     digitrow++;
193   }
194
195   /*
196 Render the second row of 2 hexadecimal digits
197 */
198   digitrow = 0; /* start at top of first row of digits to render */
199   for (row = 9; row < 14; row++) {
200     rowbits = (hexdigit[d3][digitrow] « 9) |
201               (hexdigit[d4][digitrow] « 3);
202     grid[row] ^= rowbits; /* digits appear as white on black background */
203     digitrow++;
204   }
205
206   for (row = 0; row < 16; row++) printf ("%04X", grid[row] & 0xFFFF);
207
208   putchar ('\n');
209
210   return;
211 }
```

Here is the caller graph for this function:



#### 5.14.2.2 hexprint6()

```
void hexprint6 (
                int thiscp )
```

Generate a bitmap containing a 6-digit Unicode code point.

Takes a 6-digit Unicode code point as an argument and prints a unifont.hex string for it to stdout.

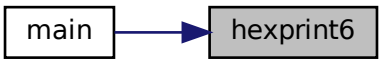
Parameters

| | | |
|---|---|---|
| in | thiscp | The current code point for which to generate a glyph. |

Definition at line 223 of file unihexgen.c.

```
224 {
225
226    int grid[16]; /* the glyph grid we'll build */
227
228    int row;      /* row number in current glyph */
229    int digitrow; /* row number in current hex digit being rendered */
230    int rowbits;  /* 1 & 0 bits to draw current glyph row */
231
232    int d1, d2, d3, d4, d5, d6; /* six hexadecimal digits of each code point */
233
234    d1 = (thiscp » 20) & 0xF;
235    d2 = (thiscp » 16) & 0xF;
236    d3 = (thiscp » 12) & 0xF;
237    d4 = (thiscp »  8) & 0xF;
238    d5 = (thiscp »  4) & 0xF;
239    d6 = (thiscp      ) & 0xF;
240
241    /* top and bottom rows are white */
242    grid[0] = grid[15] = 0x0000;
243
244    /* 14 inner rows are 16-pixel wide black lines, centered */
245    for (row = 1; row < 15; row++) grid[row] = 0xFFFF;
246
247
248    printf ("%06X:", thiscp);
249
250    /*
251 Render the first row of 3 hexadecimal digits
252 */
253    digitrow = 0; /* start at top of first row of digits to render */
254    for (row = 2; row < 7; row++) {
255       rowbits = (hexdigit[d1][digitrow] « 11) |
256                 (hexdigit[d2][digitrow] «  6) |
257                 (hexdigit[d3][digitrow] «  1);
258       grid[row] ^= rowbits; /* digits appear as white on black background */
259       digitrow++;
260    }
261
262    /*
263 Render the second row of 3 hexadecimal digits
264 */
265    digitrow = 0; /* start at top of first row of digits to render */
266    for (row = 9; row < 14; row++) {
267       rowbits = (hexdigit[d4][digitrow] « 11) |
268                 (hexdigit[d5][digitrow] «  6) |
269                 (hexdigit[d6][digitrow] «  1);
270       grid[row] ^= rowbits; /* digits appear as white on black background */
271       digitrow++;
272    }
273
274    for (row = 0; row < 16; row++) printf ("%04X", grid[row] & 0xFFFF);
275
276    putchar ('\n');
277
278    return;
279 }
```

Here is the caller graph for this function:




#### 5.14.2.3 main()

```
int main (
                int argc,
                char * argv[] )
```

The main function.

Parameters

| | | |
|---|---|---|
| in | argc | The count of command line arguments. |
| in | argv | Pointer to array of command line arguments (code point range). |

Returns

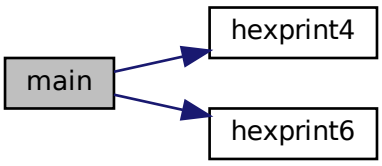
This program exits with status EXIT_SUCCESS.

Definition at line 112 of file unihexgen.c.

```
113 {
114
115    int startcp, endcp, thiscp;
116    void hexprint4(int); /* function to print one 4-digit unifont.hex code point */
117    void hexprint6(int); /* function to print one 6-digit unifont.hex code point */
118
119    if (argc != 3) {
120       fprintf (stderr,"\n%s - generate unifont.hex code points as\n", argv[0]);
121       fprintf (stderr,"four-digit hexadecimal numbers in a 2 by 2 grid,\n");
122       fprintf (stderr,"or six-digit hexadecimal numbers in a 3 by 2 grid.\n");
123       fprintf (stderr,"Syntax:\n\n");
124       fprintf (stderr,"     %s first_code_point last_code_point > glyphs.hex\n\n", argv[0]);
125       fprintf (stderr,"Example (to generate glyphs for the Private Use Area):\n\n");
126       fprintf (stderr,"     %s e000 f8ff > pua.hex\n\n", argv[0]);
127       exit (EXIT_FAILURE);
128    }
129
130    sscanf (argv[1], "%x", &startcp);
131    sscanf (argv[2], "%x", &endcp);
132
133    startcp &= 0xFFFFFF; /* limit to 6 hex digits */
134    endcp   &= 0xFFFFFF; /* limit to 6 hex digits */
135
136    /*
137 For each code point in the desired range, generate a glyph.
```

```
138 */
139   for (thiscp = startcp; thiscp <= endcp; thiscp++) {
140     if (thiscp <= 0xFFFF) {
141       hexprint4 (thiscp); /* print digits 2/line, 2 lines */
142     }
143     else {
144       hexprint6 (thiscp); /* print digits 3/line, 2 lines */
145     }
146   }
147   exit (EXIT_SUCCESS);
148 }
```

Here is the call graph for this function:




## 5.14.3 Variable Documentation

### 5.14.3.1 hexdigit

char hexdigit[16][5]

Initial value:
= {
  {0x6,0x9,0x9,0x9,0x6},
  {0x2,0x6,0x2,0x2,0x7},
  {0xF,0x1,0xF,0x8,0xF},
  {0xE,0x1,0x7,0x1,0xE},
  {0x9,0x9,0xF,0x1,0x1},
  {0xF,0x8,0xF,0x1,0xF},
  {0x6,0x8,0xE,0x9,0x6},
  {0xF,0x1,0x2,0x4,0x4},
  {0x6,0x9,0x6,0x9,0x6},
  {0x6,0x9,0x7,0x1,0x6},
  {0xF,0x9,0xF,0x9,0x9},
  {0xE,0x9,0xE,0x9,0xE},
  {0x7,0x8,0x8,0x8,0x7},
  {0xE,0x9,0x9,0x9,0xE},
  {0xF,0x8,0xE,0x8,0xF},
  {0xF,0x8,0xE,0x8,0x8}
}

Bitmap pattern for each hexadecimal digit.

hexdigit[][] definition: the bitmap pattern for each hexadecimal digit.

Each digit is drawn as a 4 wide by 5 high bitmap, so each digit row is one hexadecimal digit, and each entry has 5 rows.

For example, the entry for digit 1 is:

{0x2,0x6,0x2,0x2,0x7},

which corresponds graphically to:

–#- ==> 0010 ==> 0x2 -##- ==> 0110 ==> 0x6 –#- ==> 0010 ==> 0x2 –#- ==> 0010 ==> 0x2 -### ==> 0111 ==> 0x7

These row values will then be exclusive-ORed with four one bits (binary 1111, or 0xF) to form white digits on a black background.

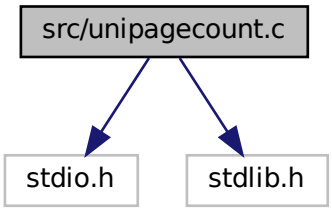
Functions hexprint4 and hexprint6 share the hexdigit array; they print four-digit and six-digit hexadecimal code points in a single glyph, respectively.

Definition at line 84 of file unihexgen.c.

## 5.15 src/unipagecount.c File Reference

unipagecount - Count the number of glyphs defined in each page of 256 code points

#include <stdio.h>
#include <stdlib.h>
Include dependency graph for unipagecount.c:




### Macros

- #define MAXBUF 256

  Maximum input line size - 1.

## Functions

- int main (int argc, char *argv[])
  The main function.
- void mkftable (unsigned plane, int pagecount[256], int links)
  Create an HTML table linked to PNG images.

### 5.15.1 Detailed Description

unipagecount - Count the number of glyphs defined in each page of 256 code points

Author

Paul Hardy, unifoundry <at> unifoundry.com, December 2007

Copyright



This program counts the number of glyphs that are defined in each "page" of 256 code points, and prints the counts in an 8 x 8 grid. Input is from stdin. Output is to stdout.

The background color of each cell in a 16-by-16 grid of 256 code points is shaded to indicate percentage coverage. Red indicates 0% coverage, green represents 100% coverage, and colors in between pure red and pure green indicate partial coverage on a scale.

Each code point range number can be a hyperlink to a PNG file for that 256-code point range's corresponding bitmap glyph image.

Synopsis:

```
unipagecount < font_file.hex > count.txt
unipagecount -phex_page_num < font_file.hex  -- just 256 points
unipagecount -h < font_file.hex              -- HTML table
unipagecount -P1 -h < font.hex > count.html  -- Plane 1, HTML out
unipagecount -l < font_file.hex              -- linked HTML table
```

### 5.15.2 Function Documentation

#### 5.15.2.1 main()

```
int main (
            int argc,
            char * argv[] )
```

The main function.

Parameters

| | | |
|---|---|---|
| in | argc | The count of command line arguments. |
| in | argv | Pointer to array of command line arguments. |

Returns

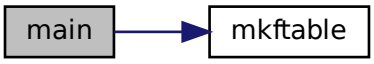
This program exits with status 0.

Definition at line 67 of file unipagecount.c.

```
68 {
69
70   char inbuf[MAXBUF]; /* Max 256 characters in an input line */
71   int i, j;  /* loop variables */
72   unsigned plane=0; /* Unicode plane number, 0 to 0x16 */
73   unsigned page;  /* unicode page (256 bytes wide) */
74   unsigned unichar; /* unicode character */
75   int pagecount[256] = {256 * 0};
76   int onepage=0; /* set to one if printing character grid for one page */
77   int pageno=0; /* page number selected if only examining one page */
78   int html=0;   /* =0: print plain text; =1: print HTML */
79   int links=0;  /* =1: print HTML links; =0: don't print links */
80   void mkftable();  /* make (print) flipped HTML table */
81
82   size_t strlen();
83
84   if (argc > 1 && argv[1][0] == '-') {  /* Parse option */
85     plane = 0;
86     for (i = 1; i < argc; i++) {
87       switch (argv[i][1]) {
88         case 'p':  /* specified -p<hexpage> -- use given page number */
89           sscanf (&argv[1][2], "%x", &pageno);
90           if (pageno >= 0 && pageno <= 255) onepage = 1;
91           break;
92         case 'h':  /* print HTML table instead of text table */
93           html = 1;
94           break;
95         case 'l':  /* print hyperlinks in HTML table */
96           links = 1;
97           html = 1;
98           break;
99         case 'P':  /* Plane number specified */
100          plane = atoi(&argv[1][2]);
101          break;
102      }
103    }
104  }
105  /*
106 Initialize pagecount to account for noncharacters.
107 */
108   if (!onepage && plane==0) {
109     pagecount[0xfd] = 32;  /* for U+FDD0..U+FDEF */
110   }
111   pagecount[0xff] = 2;   /* for U+nnFFFE, U+nnFFFF */
112   /*
113 Read one line at a time from input.  The format is:
114
115 <hexpos>:<hexbitmap>
116
117 where <hexpos> is the hexadecimal Unicode character position
118 in the range 00..FF and <hexbitmap> is the sequence of hexadecimal
119 digits of the character, laid out in a grid from left to right,
120 top to bottom.  The character is assumed to be 16 rows of variable
121 width.
122 */
123   while (fgets (inbuf, MAXBUF-1, stdin) != NULL) {
124     sscanf (inbuf, "%X", &unichar);
125     page = unichar » 8;
126     if (onepage) { /* only increment counter if this is page we want */
127       if (page == pageno) { /* character is in the page we want */
128         pagecount[unichar & 0xff]++; /* mark character as covered */
129       }
130     }
```

```
131       else { /* counting all characters in all pages */
132         if (plane == 0) {
133           /* Don't add in noncharacters (U+FDD0..U+FDEF, U+FFFE, U+FFFF) */
134           if (unichar < 0xfdd0 || (unichar > 0xfdef && unichar < 0xfffe))
135             pagecount[page]++;
136         }
137         else {
138           if ((page » 8) == plane) { /* code point is in desired plane */
139             pagecount[page & 0xFF]++;
140           }
141         }
142       }
143     }
144     if (html) {
145       mkftable (plane, pagecount, links);
146     }
147     else {  /* Otherwise, print plain text table */
148       if (plane > 0) fprintf (stdout, "  ");
149       fprintf (stdout,
150         "    0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F\n");
151       for (i=0; i<0x10; i++) {
152         fprintf (stdout,"%02X%X ", plane, i); /* row header */
153         for (j=0; j<0x10; j++) {
154           if (onepage) {
155             if (pagecount[i*16+j])
156               fprintf (stdout," *  ");
157             else
158               fprintf (stdout," .  ");
159           }
160           else {
161             fprintf (stdout, "%3X ", pagecount[i*16+j]);
162           }
163         }
164         fprintf (stdout,"\n");
165       }
166
167     }
168     exit (0);
169 }
```

Here is the call graph for this function:




#### 5.15.2.2 mkftable()

```
void mkftable (
                unsigned plane,
                int pagecount[256],
                int links )
```

Create an HTML table linked to PNG images.

This function creates an HTML table to show PNG files in a 16 by 16 grid. The background color of each "page" of 256 code points is shaded from red (for 0% coverage) to green (for 100% coverage).

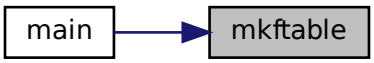
Parameters

| | | |
|---|---|---|
| in | plane | The Unicode plane, 0..17. |
| in | pagecount | Array with count of glyphs in each 256 code point range. |
| in | links | 1 = generate hyperlinks, 0 = do not generate hyperlinks. |

Definition at line 185 of file unipagecount.c.

```
186 {
187   int i, j;
188   int count;
189   unsigned bgcolor;
190
191   printf ("<html>\n");
192   printf ("<body>\n");
193   printf ("<table border=\"3\" align=\"center\">\n");
194   printf ("  <tr><th colspan=\"16\" bgcolor=\"#ffcc80\">");
195   printf ("GNU Unifont Glyphs<br>with Page Coverage for Plane %d<br>(Green=100%%, Red=0%%)</th></tr>\n", plane);
196   for (i = 0x0; i <= 0xF; i++) {
197     printf ("  <tr>\n");
198     for (j = 0x0; j <= 0xF; j++) {
199       count = pagecount[ (i << 4) | j ];
200
201       /* print link in cell if links == 1 */
202       if (plane != 0 || (i < 0xd || (i == 0xd && j < 0x8) || (i == 0xf && j > 0x8))) {
203         /* background color is light green if completely done */
204         if (count == 0x100) bgcolor = 0xccffcc;
205         /* otherwise background is a shade of yellow to orange to red */
206         else bgcolor = 0xff0000 | (count << 8) | (count >> 1);
207         printf ("    <td bgcolor=\"#%06X\">", bgcolor);
208         if (plane == 0)
209           printf ("<a href=\"png/plane%02X/uni%02X%X%X.png\">%X%X</a>", plane, plane, i, j, i, j);
210         else
211           printf ("<a href=\"png/plane%02X/uni%02X%X%X.png\">%02X%X%X</a>", plane, plane, i, j, plane, i, j);
212         printf ("</td>\n");
213       }
214       else if (i == 0xd) {
215         if (j == 0x8) {
216           printf ("    <td align=\"center\" colspan=\"8\" bgcolor=\"#cccccc\">");
217           printf ("<b>Surrogate Pairs</b>");
218           printf ("</td>\n");
219         }  /* otherwise don't print anything more columns in this row */
220       }
221       else if (i == 0xe) {
222         if (j == 0x0) {
223           printf ("    <td align=\"center\" colspan=\"16\" bgcolor=\"#cccccc\">");
224           printf ("<b>Private Use Area</b>");
225           printf ("</td>\n");
226         }  /* otherwise don't print any more columns in this row */
227       }
228       else if (i == 0xf) {
229         if (j == 0x0) {
230           printf ("    <td align=\"center\" colspan=\"9\" bgcolor=\"#cccccc\">");
231           printf ("<b>Private Use Area</b>");
232           printf ("</td>\n");
233         }
234       }
235     }
236     printf ("  </tr>\n");
237   }
238   printf ("</table>\n");
239   printf ("</body>\n");
240   printf ("</html>\n");
241
242   return;
243 }
```

Here is the caller graph for this function:



# Index